Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 61 (2006) 152–187
Science of Computer Programming
www.elsevier.com/locate/scico

# On the expressiveness of timed coordination models

I. Linden[a,*], J.-M. Jacquet[a], K. De Bosschere[b], A. Brogi[c]

[a] *Institute of Informatics, University of Namur, Namur, Belgium*
[b] *Electronics Department, Ghent University, Ghent, Belgium*
[c] *Department of Computer Science, University of Pisa, Pisa, Italy*

Received 30 January 2004; received in revised form 24 May 2005; accepted 10 October 2005

**Abstract**

Although very simple and elegant, Linda-style coordination models lack the notion of time, and are therefore not able to precisely model real-life coordination applications. Nevertheless, industrial proposals such as TSpaces and JavaSpaces, inspired from Linda, have incorporated time constructs.

This paper aims at a systematic study of the introduction of time in coordination models. It builds upon previous work to study in a coherent framework the expressiveness of Linda extended with two notions of time, relative time and absolute time, and, for each notion, two types of features. On the one hand, with respect to relative time, we describe two extensions: (i) a delay mechanism to postpone the execution of communication primitives, and (ii) explicit deadlines on the validity of tuples and on the duration of suspension of communication operations. On the other hand, for absolute time, we introduce: (iii) a wait primitive capable of waiting till an absolute point of time, and (iv) time intervals, both on tuples in the data store and on communication operations.

This expressiveness study points out a most expressive language for which an implementation is described, thereby allowing for the implementation of all the languages presented in the paper.


*Keywords:* Coordination; Time; Expressiveness; Embedding

## 1. Introduction

As motivated by the constant expansion of computer networks and illustrated by the development of distributed applications, the design of modern software systems centers on re-using and integrating software components. This induces a paradigm shift from stand-alone applications to interacting distributed systems, which, in turn, naturally calls for well-defined methodologies and tools aiming at integrating heterogeneous software components.

In order to tackle properly the development of modern software, a clear separation between the *interaction* and the *computational* aspects of software components has been advocated by Gelernter and Carriero in [26]. Their claim has been supported by the design of a model, Linda [18], originally presented as a set of inter-agent communication

* Corresponding author. Tel.: +32 81 72 49 87; fax: +32 81 72 49 67.
*E-mail addresses:* ili@info.fundp.ac.be (I. Linden), jmj@info.fundp.ac.be (J.-M. Jacquet), kdb@elis.ugent.be (K. De Bosschere), brogi@di.unipi.it (A. Brogi).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.10.011

primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing for the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, were proposed afterwards. Some of them extend Linda in different ways, for instance by introducing multiple dataspaces and meta-level control rules (e.g., Bauhaus Linda [40], Bonita [47], $\mu Log$ [33], PoliS [20], Shared Prolog [14]), by addressing open distributed systems (e.g., Laura [54]), middleware web-based environments (e.g., Jada [21]), or mobility (e.g., KLAIM [41]). A number of other coordination models rely on a notion of shared dataspace, e.g., Concurrent Constraint Programming [50], Gamma [8], Linear Objects [2] and Manifold [3], to cite only a few. A comprehensive survey of these and other coordination models and languages has been reported in [44].

However, the coding of applications evidences the fact that data rarely has an eternal life and that services have to be provided for a bounded amount of time. For instance, a request for information on the web has to be satisfied in a reasonable amount of time. Even more crucial is the request for an ambulance which not only has to be answered eventually but also within a critical period of time. The list could also be continued with software in the areas of air-traffic control, manufacturing plants and telecommunication switches, which are inherently reactive and for which interaction must occur in "real time".

Although there is an obvious application need, the introduction of time has not been deeply studied in the context of coordination languages and models, the notable exceptions being [10,43,48,49], yet proposed in the context of concurrent constraint programming, and [15,16].

Our recent work has aimed at contributing to the study of time in coordination languages and models. In [34], we have described four ways of introducing time in Linda-like languages. They rely on two notions of time, relative time and absolute time, and, for each notion, on two types of features: delay mechanisms and explicit deadlines on the validity of tuples and on the duration of suspension of communication operations. In addition to the description of the language primitives, elementary expressiveness results have been presented and implementation techniques have been detailed. In [32] and [35], respectively, a systematic study of relative time and absolute time coordination languages has been performed. This paper aims at presenting a complete picture of all these results in a coherent setting. In particular, as will be appreciated by the reader, we shall introduce a new notion of embedding making explicit use of the global clock.

Following previous work, we shall use the so-called *two-phase functioning* approach to real-time systems illustrated by languages such as Lustre [19], Esterel [9] and Statecharts [28]. This approach may be described as follows. In a first phase, elementary actions of statements are executed. They are assumed to be atomic in the sense that they take no time. Similarly, composition operators are assumed to be executed at no cost. In a second phase, when no actions can be reduced or when all the components encounter a special timed action, time progresses by one unit. Although simple, this approach has been proved to be effective for modeling reactive systems.

Related proposals for the introduction of time in coordination-like languages mainly fall in the category of relative time languages, namely languages where time is not considered with respect to time instants of a clock but rather with respect to durations. For instance, [48] introduces time in the concurrent constraint setting[1] [50] by identifying quiescent points in the computation where no new information is introduced and by providing an operator for delaying computations by one unit. At each quiescent point of time, the dataspace is reinitialized to an empty content. The paper [49] extends this framework, on the one hand, by introducing a primitive for checking the absence of information and reacting to this absence during the same unit of time and, on the other hand, by generalizing the delay mechanism in a $hence\ A$ construct which states that $A$ holds at every instant after the considered time. The resulting languages are called *tcc* and *tdcc*.

The paper [53] has shown that the language *tcc* can embed one classical representative of the state oriented synchronous languages, namely Argos [37], and one representative of the declarative class of dataflow synchronous languages, namely Lustre [19].

De Boer, Gabbrielli, and Meo have presented in [10] a timed interpretation of concurrent languages by fixing the time needed for the execution of parallel tell and ask operations as one unit and by interpreting action prefixing as the next operator. A delay mechanism is presented in Oz [52], a language which combines object oriented features

[1] Concurrent constraint languages may be viewed as a variant of Linda restricted to two communication primitives putting information on a dataspace and checking for the presence of information on it.

with symbolic computation and constraints, and (relative) time-outs have been introduced in TSpaces [55] and JavaSpaces [25]. A formal semantics of these time-outs and other mechanisms, different from our expressiveness study, is presented in [15].

Another piece of work on the expressiveness of timed constraint systems is [43]. There, various extensions of the *tcc* languages have been studied: extension with replication and recursion static scoping. Decidability results are proved as well as several encodings, which are however not of the form of modular phased embeddings studied in this paper.

The article [16] investigates the impact of various mechanisms for expired data collection on the expressiveness of coordination systems. However, the study is based on Random Access Machines, on ordered and unordered tells of timed data and on decidability results.

As may be appreciated from the above description, our work is quite different. We shall study absolute time in addition to relative time, shall examine a richer class of languages and will use to that end a new form of embedding. Moreover, we study relative expressiveness properties in contrast to [16] where absolute expressiveness properties are established.

Finally, time has been extensively studied in process algebras. Examples of these timed process algebras are [4,22, 27,29,39,42,45,46]. However, to the best of our knowledge, these pieces of work are all extensions of classical process algebras such as CCS [38], CSP [30], ACP [7] and as such incorporate a synchronous form of communication based on name sharing. In contrast, the communication between processes in the timed coordination languages we study is of an asynchronous nature and rests on the availability of information. As will be appreciated by the reader, this directly leads to different kinds of comparisons.

Moreover our work has different technical roots. For instance, [5] and its extension [6] relate several timed process algebras either by the notion of conservative extension of equality theories or by a notion of embedding based on the existence of injective mappings $h$ from the terms of an algebra $T = (\Sigma, E)$ to the terms of another algebra $T' = (\Sigma', E')$ such that, $T \vdash s = t$ implies $T' \vdash h(s) = h(t)$ for all closed terms $s$ and $t$. A similar technique of mapping is used in [17] to compare Temporal CCS [39] with closed interval process algebra [1,23]. In contrast to these pieces of work, we do not employ equality theories but use coders and decoders which not only require transitions of an agent to be simulated by its coding but also the final states reached to agree on decoding.

In the context of Abstract State Machines (ASM [12]), stepwise refinement is proposed as a practical method for crossing abstraction levels and linking ASM models through incremental development steps, starting from ground models and turning them into executable code in a stepwise fashion [11]. ASM refinement has been successfully employed to verify compiler correctness (e.g., Java to JVM, Prolog to WAM [12]). The embedding employed by ASM refinement however relies on the notion of equivalent states and does not impose constraints (such as modularity) on the coding.

The rest of this paper is structured as follows. Section 2 introduces the families of languages under study in the paper. All of them rest on common sequential, parallel and choice operators. The Linda-like languages are first modelled as the $\mathcal{L}$ family. Wait mechanisms are then introduced and absolute timing primitives are defined thereafter. Section 3 introduces the framework for comparing the expressiveness of languages. To that end, we shall refine the notion of modular embedding proposed in [24] to our time context presented in phases. This will lead to the notion of phased embedding and strong phased embedding. The expressiveness hierarchy of each family of languages, considered in isolation, is studied in Section 4. Families are then compared in Section 5. Based on these results, Section 6 sketches the design of an implementation of the languages. Finally, Section 7 draws our conclusion and draws lines for future research.

In order to keep the paper to a reasonable length, only the main ideas are given in the case of proofs which are obvious or which consist of reformulation of other proofs. However, the interested reader may consult [36] where proofs are given in detail.

## 2. The families of languages

### 2.1. Common syntax and rules

All languages considered in this paper contain sequential, parallel and choice operators. They differ only in the set of communication primitives they embody. As a result, assuming such a set, the syntax of a statement, subsequently

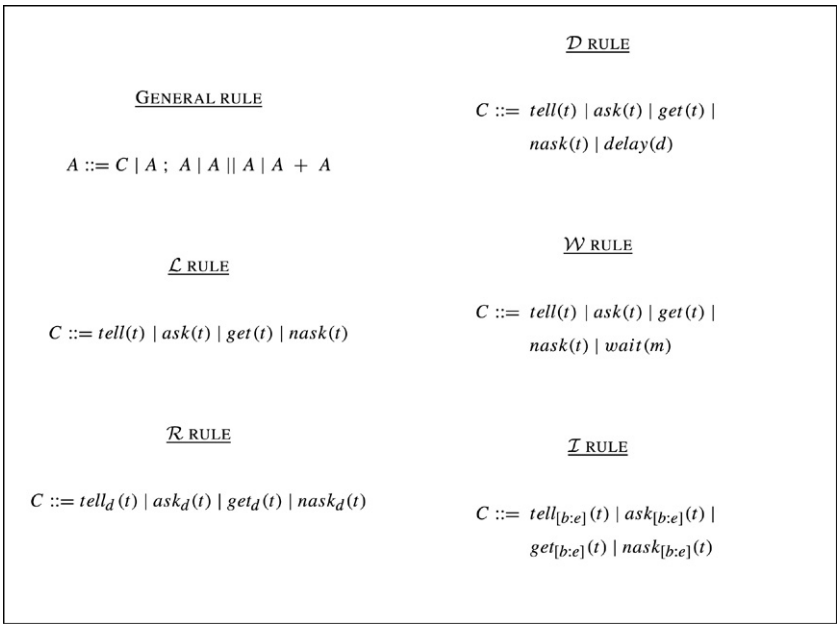


Fig. 1. Comparative syntax of the languages.

called an agent, is defined by the "general rule" of Fig. 1 and its semantics is provided by rules (S), (P), and (C) of Fig. 2. There, as we shall see later, configurations are of the form $\langle A \mid \sigma \rangle$ where $A$ represents the agent under consideration and $\sigma$ represents a memory, to be specified for each family of languages.

Note that, for simplicity of presentation, only finite processes are treated here, in view of the observation that infinite processes can be handled by extending the results of this paper in the classical way, as exemplified for instance in [31].

### 2.2. The family of Linda-like concurrent languages

To start with, consider the family of languages $\mathcal{L}(\mathcal{X})$, parameterized on the set of Linda-like communication primitives $\mathcal{X}$. This set $\mathcal{X}$ consists of the basic Linda primitives `out`, `in`, and `rd`, for putting an object in a shared dataspace, getting it and checking for its presence, respectively, together with a primitive testing for the absence of an object from the dataspace. With the aim of keeping an analogy with concurrent constraint programming, the first three primitives have been renamed as `tell`, `get`, `ask` and, accordingly, the last primitive as `nask`. Formally, the language is defined as follows,

**Definition 1.** Let *Stoken* be an enumerable set, the elements of which are subsequently called *tokens* and are typically represented by the letters $t$ and $u$. Define the set of communication actions *Scom* as the set generated by the $\mathcal{L}$ rule of Fig. 1. Moreover, for any subset $\mathcal{X}$ of *Scom*, define the language $\mathcal{L}(\mathcal{X})$ as the set of agents $A$ generated by the general rule of Fig. 1.

For any $\mathcal{X}$, computations in $\mathcal{L}(\mathcal{X})$ may be modelled by a transition system written in Plotkin's style. Following the intuition, most of the configurations consist of an agent together with a multiset of tokens denoting the tokens currently available for the computation. To easily express termination, we shall introduce particular configurations composed of a special terminating symbol $E$ together with a multiset of tokens. For uniformity, we shall abuse language and qualify $E$ as an agent. However, to meet the intuitive expectation, we shall always rewrite agents of the form $(E\ ;\ A)$, $(E \parallel A)$, and $(A \parallel E)$ as $A$. This is technically achieved by defining the extended set of agents as follows, and through simplifications derived by imposing a bimonoid structure.

**Definition 2.** Define the extended set of agents *Seagent* by the following grammar

$$A ::= E \mid C \mid A\ ;\ A \mid A \parallel A \mid A\ +\ A.$$

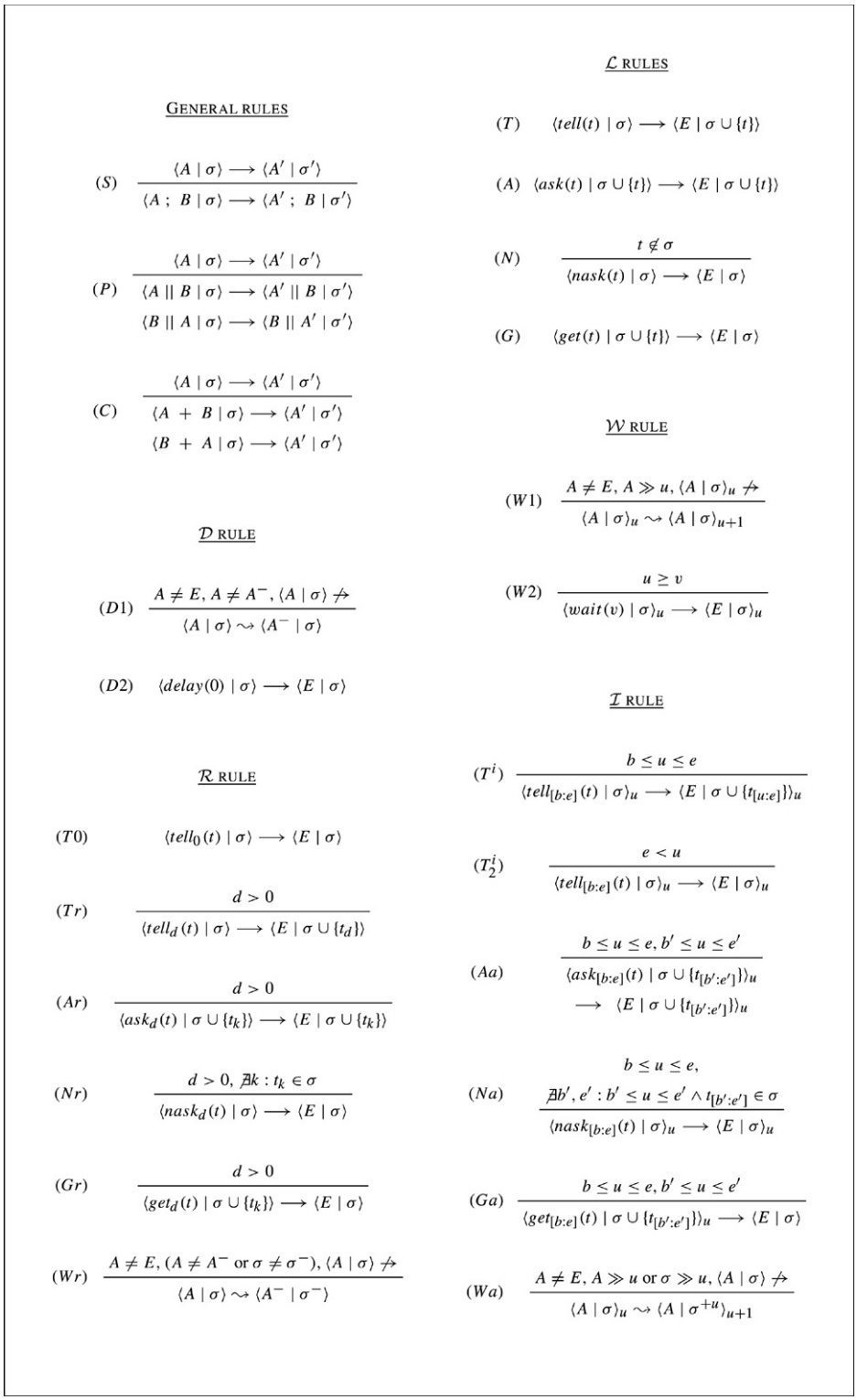


Fig. 2. Comparative semantics of the languages.

Moreover, we shall subsequently assert that the structure (*Seagent*, $E$, ; , ||) is a bimonoid and simplify elements of *Seagent* accordingly.

**Definition 3.** Define the set of stores *Sstore* as the set of finite multisets with elements from Stoken.

**Definition 4.** Define the set of configurations *Sconf* as $Seagent \times Sstore$. Configurations are denoted as $\langle A \mid \sigma \rangle$, where $A$ is an (extended) agent and $\sigma$ is a multiset of tokens.

**Definition 5.** The transition rules for the $\mathcal{L}$ agents are the general ones of Fig. 2 together with rules (T), (A), (N), (G) of that figure, where $\sigma$ denotes a multiset of tokens.

Rule (T) states that an atomic agent $tell(t)$ can be executed in any store $\sigma$, and that its execution results in adding the token $t$ to the store $\sigma$. Rules (A) and (N) state respectively that the atomic agents $ask(t)$ and $nask(t)$ can be executed in any store containing the token $t$ and not containing $t$, and that their execution does not modify the current store. Rule (G) also states that an atomic agent $get(t)$ can be executed in any store containing an occurrence of $t$, and it deletes the occurrence of $t$ from the resulting store. Note that the symbol $\cup$ actually denotes multiset union.

We are now in a position to define the operational semantics.

**Definition 6.**

(1) Let $\delta^+$ and $\delta^-$ be two fresh symbols denoting respectively success and failure. Define the set of final states *Sfstate* as the set $Sstore \times \{\delta^+, \delta^-\}$.
(2) Define the *operational semantics* $\mathcal{O} : Sagent \to \mathcal{P}(Sfstate)$ as the following function: For any agent $A$,

$$\begin{aligned}\mathcal{O}(A) \;=\; & \{(\sigma, \delta^+) : \langle A \mid \emptyset\rangle \to^* \langle E \mid \sigma\rangle\} \\ & \cup \{(\sigma, \delta^-) : \langle A \mid \emptyset\rangle \to^* \langle B \mid \sigma\rangle \not\to, B \neq E\}.\end{aligned}$$

### 2.3. Normal form

A classical result of concurrency theory is that modeling parallel composition by interleaving, as we did, allows agents to be considered in a normal form. We first define what this actually means, and then state the proposition that agents and their normal forms are equivalent in the sense that they yield the same computations.

**Definition 7.** Given a subset $\mathcal{X}$ of *Scom*, the set *Snagent* of agents in normal form is defined by the following rule, where $N$ is an agent in normal form and $c$ denotes a communication action of $\mathcal{X}$.

$$N ::= c \mid c\ ;\ N \mid N\ +\ N.$$

**Proposition 1.** *For any agent $A$, there is an agent in normal form $N$ such that $\mathcal{O}(A) = \mathcal{O}(N)$.*

### 2.4. The family of Linda-like concurrent languages with relative time

#### 2.4.1. The family of Linda-like concurrent languages with delay

One way of introducing time in coordination languages is to postpone the execution of the primitives for some period of time. This amounts to introducing a special delay primitive.

**Definition 8.** Let *Stime* be the set of positive integers. Define the set *Sdcom* as the set generated by the $\mathcal{D}$ rule of Fig. 1, where $t \in Stoken$ and $d \in Stime$. Moreover, for any subset $\mathcal{X}$ of $Sdcom \setminus \{delay\}$, define the language $\mathcal{D}(\mathcal{X})$ as the set of agents generated by the general rule of Fig. 1 for $C \in \mathcal{X} \cup \{delay\}$.

The configurations to be considered here are similar to those used for the $\mathcal{L}$ family. However, time needs to be taken into account explicitly in the transitions. This is done in two ways. First, by the introduction of a new rule (D1), which defines a new transition relation $\rightsquigarrow$ to express the progress of time by one unit. In fact, the $\to$ reduction is used to model the first phase of the two-phase functioning approach to real time while the $\rightsquigarrow$ relation is used to model the second phase of this approach.

Second, as a result of the progress of time, delays under reduction must be decreased by one unit. This is achieved by the $A^-$ construct. Note that, to avoid the computation infinitely trying to decrease blocked non-delay primitives, rule (D1) requires $A^-$ to express some progress, namely to be different from $A$.

Finally, rule (D2) is introduced to reduce a delay of 0 unit of time to $E$.

Summing up, the transitions to be considered are defined as follows.

**Definition 9.** Define the set of configurations $Sdconf$ as $Seagent' \times Sstore$, where $Seagent'$ is the set of extended agents defined as in Definition 2 but by taking $C \in Sdcom$ instead of $C \in Scom$.

**Definition 10.** Given an agent $A \in \mathcal{D}(\mathcal{X})$, we denote by $A^-$ the agent defined inductively as follows where $d > 0$

$$\begin{array}{rclrclrcl} tell(t)^- & = & tell(t) & get(t)^- & = & get(t) & (B\ ;\ C)^- & = & B^-\ ;\ C \\ ask(t)^- & = & ask(t) & delay(0)^- & = & delay(0) & (B\ ||\ C)^- & = & B^-\ ||\ C^- \\ nask(t)^- & = & nask(t) & delay(d)^- & = & delay(d-1) & (B\ +\ C)^- & = & B^-\ +\ C^- \end{array}$$

**Definition 11.** Define the transition rules for the $\mathcal{D}$ agents as the general ones of Fig. 2 and rules (T), (A), (N), (G), (D1) and (D2) of that figure.

The operational semantics is defined by integrating the two phase relations in one relation.

**Definition 12.**

(1) Let $\longmapsto$ be the relation defined by $\langle A \mid \sigma\rangle \longmapsto \langle B \mid \tau\rangle$ iff $\langle A \mid \sigma\rangle \to \langle B \mid \tau\rangle$ or $\langle A \mid \sigma\rangle \rightsquigarrow \langle B \mid \tau\rangle$.
(2) Define the *operational semantics* $\mathcal{O}_d : \mathcal{D}(Sdcom) \to \mathcal{P}(Sfstate)$ as the following function: For any timed agent $A$,

$$\begin{aligned} \mathcal{O}_d(A) = {} & \{(\sigma, \delta^+) : \langle A \mid \emptyset\rangle \longmapsto^* \langle E \mid \sigma\rangle\} \\ & \cup \{(\sigma, \delta^-) : \langle A \mid \emptyset\rangle \longmapsto^* \langle B \mid \sigma\rangle \not\longmapsto, B \neq E\}. \end{aligned}$$

*2.4.2. The family of Linda-like concurrent languages with relative durations*

A second way of introducing time in the family $\mathcal{L}(\mathcal{X})$ consists of enriching the primitives *ask*, *nask*, *get*, and *tell* themselves by durations. Formally, the new family of languages $\mathcal{R}(\mathcal{X})$ is defined as follows.

**Definition 13.** Define the set $Stcom$ of timed communication primitives as the one generated by the $\mathcal{R}$ rule of Fig. 1, where $t \in Stoken$ and $d \in Stime \cup \{\infty\}$. For any subset $\mathcal{X}$ of $Stcom$, define the language $\mathcal{R}(\mathcal{X})$ as the set of agents generated by the general rule of Fig. 1.

The configurations to be considered for the family $\mathcal{R}(\mathcal{X})$ are similar to those used for the family $\mathcal{L}(\mathcal{X})$. The introduction of time induces here the following adaptations:

(1) The intuition behind the construct $tell_d(t)$ is that $t$ is added to the store but for $d$ units of time only. To capture this fact, the tokens of the store have associated durations.
(2) As another consequence, this duration has to be updated after each tick of the clock. This motivates the introduction of the "$-$" operator acting on the store.
(3) Similarly, the intuition behind the $ask_d(t)$, $nask_d(t)$, and $get_d(t)$ primitives is that, if needed, suspension may occur only up to $d$ units of time. As a result, a similar operator, also denoted as "$-$", has to be introduced to decrease the period of suspension after each tick of the clock.

This intuition leads to the following definitions.

**Definition 14.**

(1) Given an agent $A \in \mathcal{R}(\mathcal{X})$, we denote by $A^-$ the agent defined inductively as follows[2]:

$$\begin{array}{rclrcl} tell_d(t)^- & = & tell_d(t) & & & \\ ask_d(t)^- & = & ask_{max\{0,d-1\}}(t) & (B\ ;\ C)^- & = & B^-\ ;\ C \\ nask_d(t)^- & = & nask_{max\{0,d-1\}}(t) & (B\ ||\ C)^- & = & B^-\ ||\ C^- \\ get_d(t)^- & = & get_{max\{0,d-1\}}(t) & (B\ +\ C)^- & = & B^-\ +\ C^- \end{array}$$

(2) Define the set of timed stores $Ststore$ as the set of multisets of elements of the form $t_d$ where $t$ is a token and $d$ is a duration. Given a timed store $\sigma$, we denote by $\sigma^-$ the new store obtained by decreasing the duration associated with the tokens by one unit and by removing those associated in $\sigma$ with 1 unit of time: precisely, if all the notation is understood to relate to multisets: $\sigma^- = \{t_{d-1} : t_d \in \sigma, d > 1\}$. As a simple generalization, given a strictly positive integer $n$, we denote by $\sigma^{-n}$ the store $\sigma$ updated after $n$ units of time: $\sigma^{-n} = \{t_{d-n} : t_d \in \sigma, d > n\}$.

[2] We extend classical arithmetic on natural numbers by $\infty - 1 = \infty$.

(3) Define the set of configurations *Sconf* as *Seagent* × *Ststore*. Configurations are denoted as $\langle A \mid \sigma \rangle$, where $A$ is an (extended) timed agent and $\sigma$ is a timed store.

Due to the introduction of time, the operational semantics is defined by means of the transition relations $\rightarrow$ and $\rightsquigarrow$ describing the two-phase approach. They basically adapt the relations defined for the $\mathcal{L}$ family. Accordingly, rules (Tr), (Ar), (Nr), and (Gr) adapt respectively rules (T), (A), (N), (G) in the obvious way by requiring that communication primitives be executed only for a strictly positive duration. Moreover, rule (T0) states that telling a token for a zero duration succeeds without updating the store. Rule (Wr) is the analogue of rule (D1).

**Definition 15.** Define the transition rules for the $\mathcal{R}$ agents as rules (S), (P), (C), (T0), (Tr), (Ar), (Nr), (Gr), (Wr) of Fig. 2.

The operational semantics is defined by using an auxiliary relation $\longmapsto$, defined similarly to in the previous subsection. We shall subsequently write this semantics as $\mathcal{O}_r$.

**Definition 16.** Define the *operational semantics* $\mathcal{O}_r : \mathcal{R}(Srcom) \rightarrow \mathcal{P}(Sfstate)$ as the following function: For any timed agent $A$,

$$\begin{aligned}\mathcal{O}_r(A) \;=\; & \{(\sigma^*, \delta^+) : \langle A \mid \emptyset\rangle \longmapsto^* \langle E \mid \sigma\rangle\} \\ & \cup \{(\sigma^*, \delta^-) : \langle A \mid \emptyset\rangle \longmapsto^* \langle B \mid \sigma\rangle \not\longmapsto, B \neq E\}.\end{aligned}$$

where $\sigma^*$ denotes the multiset of the tokens present in $\sigma$ without their duration.

### 2.5. The family of Linda-like concurrent languages with absolute time

#### 2.5.1. The family of Linda-like concurrent languages with wait declarations

The two previous subsections considered ways of introducing relative time in coordination languages. We consider now two ways of introducing absolute time.

One way consists of delaying the execution of the communication primitives after a precise point of time. This is obtained by introducing a primitive *wait*($m$) which forces suspension until time $m$ has been reached. Formally, the resulting family of languages is defined as follows.

**Definition 17.** Let *Stime* be the set of positive integers. Define the set $Swcom$ as the set generated by the $\mathcal{W}$ rule of Fig. 1, where $t \in$ *Stoken* and $m \in$ *Stime*. Moreover, for any subset $\mathcal{X}$ of $Swcom$, define the language $\mathcal{W}(\mathcal{X})$ as the set of agents generated by the general rule of Fig. 1 for $C \in \mathcal{X} \cup \{wait\}$.

The configurations to be used here are similar to those used for the $\mathcal{L}$ family of languages. Time introduced in an absolute way induces just one adaptation: to explicitly introduce time in the configurations. We are thus led to configurations of the form $\langle A \mid \sigma\rangle_u$ where $u$ represents the current time. The general rules (S), (P), (C) need then to be rephrased in this new notation. Of course, they leave the $u$ subscript unchanged. Rule (W1) is introduced to make time progress and rule (W2) is used to reduce a wait declaration. Note that time is allowed to progress only if the new situation differs from the old one. This is expressed by the relation $A \gg u$, which states that $A$ contains a *wait*($m$) operation with $m > u$.

To formally capture this notion, we start by defining $\mathcal{F}(A)$ as the set of the primitives which may be executed in a first step of $A$.

**Definition 18.** Define $\mathcal{F} : \mathcal{W}(Swcom) \rightarrow \mathcal{P}(Swcom)$ as follows: for any communication primitive $p$ and agents $A$ and $B$,

$$\begin{aligned}\mathcal{F}(p) &= \{p\} & \mathcal{F}(A \;+\; B) &= \mathcal{F}(A) \cup \mathcal{F}(B) \\ \mathcal{F}(A \;;\; B) &= \mathcal{F}(A) & \mathcal{F}(A \;||\; B) &= \mathcal{F}(A) \cup \mathcal{F}(B).\end{aligned}$$

**Definition 19.** For any agent $A$ and time $u$, $A \gg u$ holds iff there is some $m > u$ such that *wait*($m$) $\in \mathcal{F}(A)$.

**Definition 20.** Define the set of configurations $Swconf$ as the set $(\mathcal{W}(Swcom) \cup \{E\}) \times Sstore \times Stime$. Define the transition rules for the $\mathcal{W}$ agents as the general ones of Fig. 2 and rules (T), (A), (N), (G), (W1) and (W2) of that figure.

With the slight adaptation of the subscripts, the operational semantics is defined in a way similar to the semantics $\mathcal{O}$. It is subsequently noted as $\mathcal{O}_w$.

**Definition 21.** Define the operational semantics $\mathcal{O}_w : \mathcal{W}(Swcom) \to \mathcal{P}(Sfstate)$ as the following function: for any agent $A$

$$\mathcal{O}_w(A) = \{(\sigma, \delta^+) : \langle A \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma\rangle_u\} \cup \{(\sigma, \delta^-) : \langle A \mid \emptyset\rangle_1 \mapsto^* \langle B \mid \sigma\rangle_u \not\mapsto, B \neq E\}.$$

*2.5.2. The family of Linda-like concurrent languages with time intervals*

Time may also be introduced by extending the Linda primitives with time intervals during which the reduction should take place. We are thus led to communication primitives of the form $tell_{[b:e]}(t)$, $ask_{[b:e]}(t)$, $nask_{[b:e]}(t)$, $get_{[b:e]}(t)$, where we assume that $0 \leq b \leq e$. The resulting family of languages is referred to as $\mathcal{I}(\mathcal{X})$.

**Definition 22.** Define *Sicom* as the set generated by the $\mathcal{I}$ rule of Fig. 1, where $t \in Stoken$ and $b \in Stime$, $e \in Stime \cup \{\infty\}$ with $b \leq e$. Moreover, for any subset $\mathcal{X}$ of *Sicom*, define $\mathcal{I}(\mathcal{X})$ as the set of agents generated from *Sicom* by the general rule of Fig. 1.

**Definition 23.**

(1) Define the set of interval stores *Sistore* as the set of multisets of elements of the form $t_{[b:e]}$ where $t \in Stoken$, $b \in Stime$, $e \in Stime \cup \{\infty\}$ are such that $b \leq e$.
(2) Define the set of configurations $Siconf$ as the set $(\mathcal{I}(Sicom) \cup \{E\}) \times Sistore \times Stime$.
(3) For any agent $A$ of $\mathcal{I}(Sicom)$ and time $u$, define $A \gg u$ to hold if $\mathcal{F}(A)$, defined in a way similar to that of Definition 18, contains at least one primitive $tell_{[b:e]}(t)$, $ask_{[b:e]}(t)$, $nask_{[b:e]}(t)$, $get_{[b:e]}(t)$, with $b > u$.
(4) For any interval store $\sigma$ and time $u$, define $\sigma \gg u$ to hold if there is $t_{[b:e]} \in \sigma$ such that $e \neq \infty$ and $e > u$. Moreover, define $\sigma^{+u}$ as

$$\sigma^{+u} = \{t_{[max\{b,u+1\}:e]} : t_{[b:e]} \in \sigma, u + 1 \leq e\}.$$

(5) Define the set of transition rules for the $\mathcal{I}$ agents as rules (S), (P), (C) rewritten so as to include the $u$ subscript and rules (Ta), (Aa), (Na), (Ga), and (Wa) of Fig. 2.

The operational semantics is adapted from that of the previous section. It is subsequently written as $\mathcal{O}_i$.

**Definition 24.** Define the operational semantics $\mathcal{O}_i : \mathcal{I}(Sicom) \to \mathcal{P}(Sfstate)$ as the following function: for any agent $A$

$$\mathcal{O}_i(A) = \{(\sigma^*, \delta^+) : \langle A \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma\rangle_u\} \cup \{(\sigma^*, \delta^-) : \langle A \mid \emptyset\rangle_1 \mapsto^* \langle B \mid \sigma\rangle_u \not\mapsto, B \neq E\}$$

where $\sigma^*$ denotes the multiset of the tokens present in $\sigma$ without their subscripted interval.

## 3. Language comparison

### *3.1. Introduction*

A natural question to ask is whether the time extensions we just introduced strictly increase the expressivity of the Linda language and, if so, whether some of the timed primitives may be expressed in terms of others.

A basic approach for answering that question has been given by Shapiro in [51] as follows. Consider two languages $L$ and $L'$. Assume given the semantics mappings (*observation criteria*) $\mathcal{S}em : L \to Obs$ and $\mathcal{S}em' : L' \to Obs'$, where $Obs$ and $Obs'$ are some suitable domains. Then, according to [51], $L$ can *embed* $L'$ if there exists a mapping $\mathcal{C}$ (*coder*) from the statements of $L'$ to the statements of $L$, and a mapping $\mathcal{D}^e$ (*decoder*) from $Obs$ to $Obs'$, such that the diagram of Fig. 3 commutes, namely such that $\mathcal{D}^e(\mathcal{S}em(\mathcal{C}(A))) = \mathcal{S}em'(A)$, for every statement $A \in L'$.

This approach is however too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. To circumvent this problem, De Boer and Palamidessi have proposed in [24] to add three

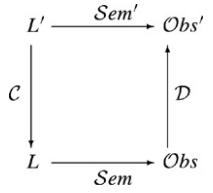


Fig. 3. Basic embedding.

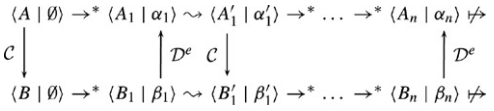


Fig. 4. Phased embedding.

constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}^e$. First, $\mathcal{D}^e$ should be defined in an element-wise way w.r.t. *Obs*:

$$\forall X \in Obs : \ \mathcal{D}^e(X) = \{\mathcal{D}^e_{el}(x) \mid x \in X\} \tag{$P_1$}$$

for some appropriate mapping $\mathcal{D}^e_{el}$. Second, the coder $\mathcal{C}$ should be defined in a compositional way w.r.t. the sequential, parallel and choice operators[3]:

$$\begin{aligned} \mathcal{C}(A \ ; \ B) &= \mathcal{C}(A) \ ; \ \mathcal{C}(B) \\ \mathcal{C}(A \ || \ B) &= \mathcal{C}(A) \ || \ \mathcal{C}(B) \\ \mathcal{C}(A \ + \ B) &= \mathcal{C}(A) \ + \ \mathcal{C}(B). \end{aligned} \tag{$P_2$}$$

Finally, the embedding should preserve the behavior of the original processes w.r.t. deadlock, failure and success (*termination invariance*):

$$\forall X \in Obs, \forall x \in X : \ tm'(\mathcal{D}^e_{el}(x)) = tm(x) \tag{$P_3$}$$

where $tm$ and $tm'$ extract the information on termination from the observables of $L$ and $L'$, respectively. An embedding satisfying these properties ($P_1$, $P_2$, $P_3$) is said to be *modular*.

### 3.2. Phased embedding

In our time context, we introduce an additional requirement associated with time. Intuitively, we require that statements and their codings agree after each phase, namely that they obey the commuting equation $\mathcal{D}^e(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$ after each phase, thus giving rise to the situation depicted in Fig. 4. A modular embedding satisfying this constraint is called modular phased embedding. The formal definition is as follows. It is phrased directly in our time coordination setting.

**Definition 25.** Define the semantics $\mathcal{O}^*$ as a generalization of the semantics $\mathcal{O}$, $\mathcal{O}_d$, $\mathcal{O}_r$, $\mathcal{O}_w$ and $\mathcal{O}_i$ to arbitrary starting store but restricted to one phase: for any agent $A$ and any store $\alpha$,

$$\begin{aligned} \mathcal{O}^*(A)(\alpha) = \ & \{(\sigma, \delta^+) : \langle A \mid \alpha\rangle \to^* \langle E \mid \sigma\rangle\} \\ & \cup \{(\sigma, \delta^-) : \langle A \mid \alpha\rangle \to^* \langle A' \mid \sigma\rangle \not\to, A' \neq E\}. \end{aligned}$$

**Definition 26.** For any agents $A$ and $B$ and any stores $\alpha$ and $\beta$, $\langle B \mid \beta\rangle$ is decodable in $\langle A \mid \alpha\rangle$ iff

(1) $\mathcal{C}(A) = B$ where the coder $\mathcal{C}$ is extended with $\mathcal{C}(E) = E$.
(2) $\mathcal{D}^e_{el}((\beta, \delta)) = (\alpha, \delta)$ where $\delta = \begin{cases} \delta^+ \text{ if } A = E = B \\ \delta^- \text{ otherwise .} \end{cases}$

[3] Actually, this is only required for the parallel and choice operators in [24].

**Definition 27.** Assume a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$. For any agents $A$, $B$, any store $\alpha$, $\beta$, $(A, \alpha)$ is *phase-simulable* in $(B, \beta)$ iff the following properties hold:

(1) $\langle B \mid \beta\rangle$ is decodable in $\langle A \mid \alpha\rangle$,
(2) for any agent $A_1$ and any store $\alpha_1$ such that $\langle A \mid \alpha\rangle \rightarrow^* \langle A_1 \mid \alpha_1\rangle \not\rightarrow$, there exist $B_1$ and $\beta_1$ such that
  (a) $\langle B \mid \beta\rangle \rightarrow^* \langle B_1 \mid \beta_1\rangle \not\rightarrow$ and $\langle B_1 \mid \beta_1\rangle$ is decodable in $\langle A_1 \mid \alpha_1\rangle$,
  (b) $\langle A_1 \mid \alpha_1\rangle \rightsquigarrow \langle A_1' \mid \alpha_1'\rangle$ iff $\langle B_1 \mid \beta_1\rangle \rightsquigarrow \langle B_1' \mid \beta_1'\rangle$, in which case $(A_1', \alpha_1')$ is phase-simulable in $(B_1', \beta_1')$,
(3) for any agent $B_1$ and any store $\beta_1$ such that $\langle B \mid \beta\rangle \rightarrow^* \langle B_1 \mid \beta_1\rangle \not\rightarrow$, there exist $A_1$ and $\alpha_1$ such that
  (a) $\langle A \mid \alpha\rangle \rightarrow^* \langle A_1 \mid \alpha_1\rangle \not\rightarrow$ and $\langle B_1 \mid \beta_1\rangle$ is decodable in $\langle A_1 \mid \alpha_1\rangle$,
  (b) $\langle A_1 \mid \alpha_1\rangle \rightsquigarrow \langle A_1' \mid \alpha_1'\rangle$ iff $\langle B_1 \mid \beta_1\rangle \rightsquigarrow \langle B_1' \mid \beta_1'\rangle$, in which case, $(A_1', \alpha_1')$ is phase-simulable in $(B_1', \beta_1')$.

**Definition 28** (*Modular Phased Embedding*). Let $L$ and $L'$ be two languages of the families $\mathcal{L}$, $\mathcal{D}$, $\mathcal{R}$, $\mathcal{W}$ and $\mathcal{I}$ and let $\mathcal{O}_x$ and $\mathcal{O}_x'$ denote their corresponding operational semantics. The language $L$ can embed $L'$ in a modular and phased manner iff there exists a coder $\mathcal{C}$ (*coder*) from the statements of $L'$ to the statements of $L$, and a decoder $\mathcal{D}^e$ (*decoder*) from $\mathcal{O}_x$ to $\mathcal{O}_x'$ such that properties $(P_1)$, $(P_2)$, $(P_3)$ hold and such that for any agent $A$ of $L'$, $(A, \emptyset)$ is phase-simulable in $(\mathcal{C}(A), \emptyset)$.

The existence of a modular phased embedding from $L'$ into $L$ is subsequently denoted by $L' \leq L$. It is easy to see that $\leq$ is a pre-order relation. Moreover if $L' \subseteq L$ then $L' \leq L$, that is, any language embeds all its sublanguages. This property descends immediately from the definition of modular phased embedding, by setting $\mathcal{C}$ and $\mathcal{D}^e$ equal to the identity function.

A stronger notion of embedding will also be employed later. The basic idea is that if two agents are independent, namely if their parallel composition executes without interaction, the codings of these agents have to enjoy the same property. We will name such embeddings strong modular phased embeddings and denote by $\leq_s$ the corresponding pre-order relation.

The notion of computations without interaction is formalized as follows.

**Definition 29.** Two agents $A$ and $B$ are said to admit a parallel computation without interaction iff there are $\sigma, \tau$ stores, $t$ and $u$ times, such that the following five properties hold where $\sigma^*$ denotes the multiset of the tokens in $\sigma$ without their subscript.

$$\begin{array}{l}
\langle A \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma\rangle_t \\
\langle B \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \tau\rangle_u \\
\sigma^* \cap \tau^* = \emptyset \\
\langle A \mid\mid B \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma^{-(u-t)} \cup \tau\rangle_u \quad \text{if } t \leq u \\
\langle A \mid\mid B \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma \cup \tau^{-(t-u)}\rangle_t \quad \text{if } t > u.
\end{array}$$

Note that in Definition 29, $t$ and $u$ respectively denote the time at which one computation of the two agents executed alone finishes. The selected parallel computation of the agents ends at the greatest of these two times.

Strong modular phased embeddings are those embeddings that admit a coder and a decoder preserving parallel computations without interaction. Formally, this is defined as follows.

**Definition 30.** Let $L$ and $L'$ be two languages and let $\mathcal{O}_x$ and $\mathcal{O}_x'$ denote their corresponding operational semantics. Let $\mathcal{C}$ be a coder from the statements of $L'$ to the statements of $L$, and $\mathcal{D}^e$ a decoder from $\mathcal{O}_x$ to $\mathcal{O}_x'$. The coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ are said to preserve parallel computations without interaction iff the following property holds. For any agents $A$ and $B$ admitting a parallel computation without interaction, and any stores $\sigma_1$ and $\tau_1$ such that

$$\begin{array}{l}
\langle \mathcal{C}(A) \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma_1\rangle_t \quad \text{with } \mathcal{D}^e_{el}((\sigma_1, \delta^+)) = (\sigma, \delta^+) \\
\langle \mathcal{C}(B) \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \tau_1\rangle_u \quad \text{with } \mathcal{D}^e_{el}((\tau_1, \delta^+)) = (\tau, \delta^+)
\end{array}$$

where $\sigma$, $\tau$, $t$ and $u$ are the stores and times employed in Definition 29, one has

$$\sigma_1^* \cap \tau_1^* = \emptyset$$

and

$$\begin{array}{l}
\langle \mathcal{C}(A \mid\mid B) \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma_1^{-(u-t)} \cup \tau_1\rangle_u \quad \text{if } t \leq u \\
\langle \mathcal{C}(A \mid\mid B) \mid \emptyset\rangle_1 \mapsto^* \langle E \mid \sigma_1 \cup \tau_1^{-(t-u)}\rangle_t \quad \text{if } t > u.
\end{array}$$

Note that if the coder and decoder are provided by Definition 28 the fact that an agent and its coder admit computations finishing at the same time is ensured by the modular phase-simulable relation between the agent and its coding.

**Definition 31** (*Strong Modular Phased Embedding*). Let $L$ and $L'$ be two languages and let $\mathcal{O}_x$ and $\mathcal{O}'_x$ denote their corresponding operational semantics. The language $L$ can embed $L'$ in a strong modular and phased manner iff there exist a coder $\mathcal{C}$ (*coder*) from the statements of $L'$ to the statements of $L$, and a decoder $\mathcal{D}^e$ (*decoder*) from $\mathcal{O}_x$ to $\mathcal{O}'_x$ such that

(1) properties $(P_1)$, $(P_2)$, $(P_3)$ hold,
(2) for any agent $A$ of $L'$, $(A, \emptyset)$ is phase-simulable in $(\mathcal{C}(A), \emptyset)$,
(3) $\mathcal{C}$ and $\mathcal{D}^e$ preserve parallel computations without interaction. $(P_4)$

### 3.3. Language analysis

The rest of the paper is devoted to the study of the expressiveness of the four families of time languages. Each of them contains 16 languages. However, except for the exhaustivity of the study, languages that do not contain the *tell* primitive are of no interest. Indeed they do not allow agents to provide information but only to consult the empty store. Therefore, we shall subsequently consider only those languages containing the *tell* primitive.

The expressiveness analysis is performed as follows. First, each family is considered in isolation in Section 4. Then, the expressiveness of the families with relative time is examined. This is followed by a study of the expressiveness of the families with absolute time. Finally, the most expressive relative and absolute time languages are compared.

For these analyses, we shall essentially use modular phased embeddings. Two results will require strong modular phased embeddings. Nevertheless, most of the results transpose to strong modular phased embeddings. Indeed, if they are separation results, then it follows directly from Definition 31 that $\mathcal{L}_1 \not\leq \mathcal{L}_2$ implies $\mathcal{L}_1 \not\leq_s \mathcal{L}_2$. If they are embedding results then, as easily checked by the reader, the coder and decoder preserve parallel computations without interactions.

## 4. Intra-family comparisons

### 4.1. The hierarchy of the languages with delay

We first turn to the $\mathcal{D}$ family of languages. A first result is that any language embeds all its sublanguages.

**Proposition 2.** *For all subsets $X$ and $Y$ of $\{ask, nask, get, tell\}$ such that $X \subseteq Y$, one has $\mathcal{D}(X) \leq \mathcal{D}(Y)$.*

We now consider the languages $\mathcal{D}(ask, tell)$ and $\mathcal{D}(nask, tell)$ obtained by extending $\mathcal{D}(tell)$ with the ability of checking the presence and the absence of data, respectively, in the dataspace. It is easy to establish that both $\mathcal{D}(ask, tell)$ and $\mathcal{D}(nask, tell)$ are strictly more expressive than $\mathcal{D}(tell)$.

**Proposition 3.** *$\mathcal{D}(ask, tell) \not\leq \mathcal{D}(tell)$ and $\mathcal{D}(nask, tell) \not\leq \mathcal{D}(tell)$.*

**Proof.** To prove the first relation, consider the agent $ask(t)$. The semantics of this agent is $\mathcal{O}_d(ask(t)) = \{(\emptyset, \delta^-)\}$. As any agent in $\mathcal{D}(tell)$ has only successful computations, it is impossible to provide a coder and a decoder satisfying property $P_3$.

The second relation is established similarly, considering the agent $tell(t); nask(t)$ whose semantics is $\mathcal{O}_d(tell(t); nask(t)) = \{(\{t\}, \delta^-)\}$. □

While $\mathcal{D}(ask, tell)$ and $\mathcal{D}(nask, tell)$ are both strictly more powerful than $\mathcal{D}(tell)$, they are not comparable to one another.

**Proposition 4.** *$\mathcal{D}(ask, tell) \not\leq \mathcal{D}(nask, tell)$ and $\mathcal{D}(nask, tell) \not\leq \mathcal{D}(ask, tell)$.*

**Proof.** To prove the first relation we proceed by contradiction. We assume that $\mathcal{D}(ask, tell) \leq \mathcal{D}(nask, tell)$ and that the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ satisfy properties $P_1$ to $P_3$ and the phased embedding property. The proof is based on the examination of the normal form of the coding of the primitive *ask*.

As $\mathcal{C}(ask(t))$ is in $\mathcal{D}(nask, tell)$, its normal form can be written as

$$\begin{aligned}\mathcal{C}(ask(t)) = {} & (delay(j_1)\ ;\ A_1) + \cdots + (delay(j_n)\ ;\ A_n) \\ & + (nask(t_1)\ ;\ B_1) + \cdots + (nask(t_m)\ ;\ B_m) \\ & + (tell(s_1)\ ;\ C_1) + \cdots + (tell(s_l)\ ;\ C_l)\end{aligned}$$

for some times $j_i$ and tokens $t_i$ and $s_i$, with $n, m, l \geq 0$.

Our first observation is that the coding cannot contain any choice starting with a *nask*, a *tell* or a *delay*(0) primitive. Indeed, if there is one choice starting with a *nask* primitive, then the coding of the agent $delay(0) + ask(t)$ accepts the following derivation

$$\langle \mathcal{C}(delay(0) + ask(t)) \mid \emptyset \rangle_1 \rightarrow \langle B_1 \mid \emptyset \rangle_1.$$

As $ask(t)$ fails on the empty store, the agent $B_1$ has to fail. This derivation provides then a valid prefix for a failing derivation of the agent. This contradicts, by property $P_3$, the fact that $delay(0) + ask(t)$ has only successful computations. The absence of choice starting with a *tell* and *delay*(0) primitive can be shown similarly.

Consequently, the agent $\mathcal{C}(ask(t))$ has then a normal form of the following type:

$$\mathcal{C}(ask(t)) = (delay(j_1)\ ;\ A_1) + \cdots + (delay(j_n)\ ;\ A_n)$$

where $j_k > 0$ $(1 \leq k \leq n)$.

A second observation about $\mathcal{C}(ask(t))$ is that, following its normal form, its computation starts with a temporal transition. This contradicts the phased embedding property. Indeed, as $ask(t)$, the agent $\mathcal{C}(ask(t))$ has to fail at time 1 without any temporal transition.

The proof of the second relation also proceeds by contradiction by considering $nask(t)$ and its coding. As before, it may be proved that the normal form of $\mathcal{C}(nask(t))$ cannot contain tell primitives. Therefore, by identifying the agent and its normal form, $\mathcal{C}(nask(t))$ is of the following type

$$\begin{aligned}\mathcal{C}(nask(t)) = {} & (delay(j_1)\ ;\ A_1) + \cdots + (delay(j_n)\ ;\ A_n) \\ & + (ask(t_1)\ ;\ B_1) + \cdots + (ask(t_m)\ ;\ B_m)\end{aligned}$$

for some times $j_i$ and tokens $t_i$, with $n, l \geq 0$ and $j_k > 0$ $(1 \leq k \leq n)$.

Consequently, the computation of $\mathcal{C}(nask(t))$ on the empty store starts with a temporal transition. However, this contradicts the phased embedding property since, as $nask(t)$, $\mathcal{C}(nask(t))$ has to succeed at time 1 without any temporal transition. □

The language $\mathcal{D}(get, tell)$ happens to be strictly more expressive than $\mathcal{D}(ask, tell)$.

**Proposition 5.** *$\mathcal{D}(ask, tell) \leq \mathcal{D}(get, tell)$ and $\mathcal{D}(get, tell) \not\leq \mathcal{D}(ask, tell)$.*

**Proof.** (i) To prove $\mathcal{D}(ask, tell) \leq \mathcal{D}(get, tell)$, we consider the coder $\mathcal{C}$ defined as follows

$$\begin{aligned}\mathcal{C}(delay(i)) &= delay(i) \\ \mathcal{C}(tell(t)) &= tell(t) \\ \mathcal{C}(ask(t)) &= get(t)\ ;\ tell(t)\end{aligned}$$

and the identity as decoder.

The coder and decoder satisfy by construction properties $P_1$ to $P_3$. The verification of the phased embedding property for any agent is established by induction on the length of the derivations.

(ii) Assume that $\mathcal{D}(get, tell) \leq \mathcal{D}(ask, tell)$ and consider $tell(a)\ ;\ get(a)$. Since $\mathcal{C}$ is compositional and since $\mathcal{O}_d(tell(a)\ ;\ get(a)) = \{(\emptyset, \delta^+)\}$, the termination mark of any element of $\mathcal{O}_d(\mathcal{C}(tell(a))\ ;\ \mathcal{C}(get(a)))$ is successful. As $\mathcal{C}(get(a))$ is composed of *ask*, *tell* and *delay* primitives only and since *ask*, *tell* and *delay* primitives do not destroy elements, it follows that any element of $\mathcal{O}_d(\mathcal{C}(tell(a))\ ;\ \mathcal{C}(get(a))\ ;\ \mathcal{C}(get(a)))$ has a successful termination mark. However, $\mathcal{O}_d(tell(a)\ ;\ get(a)\ ;\ get(a)) = \{(\emptyset, \delta^-)\}$ which contradicts property $P_3$. □

In the presence of *tell* and $get$ primitives, the *ask* primitive is redundant.

**Proposition 6.**
(i) $\mathcal{D}(get, tell) \equiv \mathcal{D}(ask, get, tell)$
(ii) $\mathcal{D}(nask, get, tell) \equiv \mathcal{D}(ask, nask, get, tell)$.

**Proof.** (i) The inequality $\mathcal{D}(get, tell) \leq \mathcal{D}(ask, get, tell)$ is obvious. The converse inequality is obtained by extending the coder of the proof of Proposition 5(i) by $\mathcal{C}(get(t)) = get(t)$. (ii) The inequality $\mathcal{D}(nask, get, tell) \leq \mathcal{D}(ask, nask, get, tell)$ is immediate. To establish the converse inequality, we first code any token $t$ by a pair of tokens which we denote as $(t_1, t_2)$. Note that this can be done because *Stoken* is enumerable: for instance, it is sufficient to associate the token associated with the integer $n$ in the enumeration with the tokens associated with the integers $2n$ and $2(n+1)$. Given such a coding of tokens, we define the coder $\mathcal{C}$ as follows.

$$\begin{array}{ll} \mathcal{C}(ask(t)) = get(t_2)\ ;\ tell(t_2) & \\ \mathcal{C}(nask(t)) = nask(t_1) & \mathcal{C}(tell(t)) = tell(t_1)\ ;\ tell(t_2) \\ \mathcal{C}(get(t)) = get(t_2)\ ;\ get(t_1) & \mathcal{C}(delay(n)) = delay(n). \end{array}$$

Moreover, the decoder $\mathcal{D}^e$ is defined as follows: $\mathcal{D}^e_{el}((\sigma, \delta)) = (\overline{\sigma}, \delta)$ where $\overline{\sigma}$ is composed of the tokens $t$ for which $t_1$ and $t_2$ are in $\sigma$, the multiplicity of occurrences of $t$ being that of pairs $(t_1, t_2)$ in $\sigma$. □

For completeness, we show now that, in the presence of the *tell* primitive, *nask* and $get$ are incomparable. The following lemma will help us in this task.

**Lemma 1.** *For any agent $A$ in $\mathcal{D}(ask, nask, tell)$, if $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \sigma \cup \tau\rangle$ then $\langle A \parallel A \mid \sigma\rangle \mapsto^* \langle B \parallel B \mid \sigma \cup \tau \cup \tau\rangle$ where $\cup$ denotes union on multisets.*

**Proof.** The proof is conducted by induction on the number of steps of the computation $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \sigma \cup \tau\rangle$. □

**Proposition 7.** *$\mathcal{D}(nask, tell) \not\leq \mathcal{D}(get, tell)$ and $\mathcal{D}(get, tell) \not\leq \mathcal{D}(ask, nask, tell)$.*

**Proof.** For the first relation, the proof is similar to that of Proposition 4. For the second relation, let us proceed by contradiction and assume that $\mathcal{D}(get, tell) \leq \mathcal{D}(ask, nask, tell)$. In that case, as $\mathcal{O}_d(tell(t)\ ;\ get(t)) = \{(\emptyset, \delta^+)\}$, any computation of $A = \mathcal{C}(tell(t))\ ;\ \mathcal{C}(get(t))$ starting in the empty store is successful by $P_3$. By Lemma 1, there is a computation of $B = \mathcal{C}(tell(t))\ ;\ (\mathcal{C}(get(t)) \parallel \mathcal{C}(get(t)))$ starting in the empty store that is successful, which contradicts, by $P_2$ and $P_3$, the fact that $\mathcal{O}_d(tell(t)\ ;\ (get(t) \parallel get(t))) = \{(\emptyset, \delta^-)\}$. □

### 4.2. The hierarchy of the languages with relative duration

As expected, the first result for the $\mathcal{R}$ family of languages is that any language embeds all its sublanguages.

**Proposition 8.** *For all subsets $X$ and $Y$ of $\{ask, nask, get, tell\}$ such that $X \subseteq Y$, one has $\mathcal{R}(X) \leq \mathcal{R}(Y)$.*

We now consider the languages $\mathcal{R}(ask, tell)$ and $\mathcal{R}(nask, tell)$ obtained by extending $\mathcal{R}(tell)$ with the ability of checking the presence and the absence of data, respectively, in the dataspace. It is easy to establish that both $\mathcal{R}(ask, tell)$ and $\mathcal{R}(nask, tell)$ are strictly more expressive than $\mathcal{R}(tell)$.

**Proposition 9.** *$\mathcal{R}(ask, tell) \not\leq \mathcal{R}(tell)$ and $\mathcal{R}(nask, tell) \not\leq \mathcal{R}(tell)$.*

**Proof.** To prove the first relation, consider the agent $ask(t)$. The semantics of this agent is $\mathcal{O}_r(ask_1(t)) = \{(\emptyset, \delta^-)\}$. As any agent in $\mathcal{R}(tell)$ has only successful computations, it is impossible to provide a coder and a decoder satisfying property $P_3$.

The second relation is established similarly by considering the agent $tell_\infty(t); nask_1(t)$ whose semantics is $\mathcal{O}_r(tell_\infty(t); nask_1(t)) = \{(\{t\}, \delta^-)\}$. □

While $\mathcal{R}(ask, tell)$ and $\mathcal{R}(nask, tell)$ are both strictly more powerful than $\mathcal{R}(tell)$, they are not comparable to one another.

**Proposition 10.** *$\mathcal{R}(ask, tell) \not\leq \mathcal{R}(nask, tell)$ and $\mathcal{R}(nask, tell) \not\leq \mathcal{R}(ask, tell)$.*

**Proof.** To prove the first relation we proceed by contradiction. We assume that $\mathcal{R}(ask, tell) \leq \mathcal{R}(nask, tell)$ and that the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ satisfy properties $P_1$ to $P_3$ and the phased embedding property. The proof is based on the examination of the normal form of the coding of the primitive *ask*.

As $\mathcal{C}(ask_1(t))$ is in $\mathcal{R}(nask, tell)$, its normal form can be written as

$$\begin{aligned}\mathcal{C}(ask_1(t)) = {} & (nask_{d_1}(t_1)\ ;\ B_1) + \cdots + (nask_{d_m}(t_m)\ ;\ B_m) \\ & + (tell_{e_1}(s_1)\ ;\ C_1) + \cdots + (tell_{e_l}(s_l)\ ;\ C_l)\end{aligned}$$

for some durations $d_i$ and $e_i$ and tokens $t_i$ and $s_i$, with $m, l \geq 0$.

Let us first observe that the coding cannot contain any choice starting with a $nask_d$ primitive with $d > 0$, or a *tell* primitive, i.e. $l = 0$ and $d_k = 0$ $(1 \leq k \leq m)$. Indeed, if there is one choice starting with a *tell* primitive, then the coding of the agent $tell_1(v) + ask_1(t)$ accepts the following derivation

$$\langle \mathcal{C}(tell_1(t) + ask_1(t)) \mid \emptyset \rangle_1 \rightarrow \langle C_1 \mid \{s_1\} \rangle_1.$$

As $ask_1(t)$ fails on the empty store, the agent $C_1$ has to fail. This derivation then provides a valid prefix for a failing derivation of the agent. This contradicts, by property $P_3$, the fact that $tell_1(v) + ask_1(t)$ has only successful computations. The absence of choice starting with a $nask_d$ primitive with $d > 0$ can be shown similarly.

Consequently, the agent $\mathcal{C}(ask_1(t))$ has a normal form of the following type:

$$\mathcal{C}(ask_1(t)) = (nask_0(t_1)\ ;\ B_1) + \cdots + (nask_0(t_m)\ ;\ B_m).$$

It follows that the computation of $\mathcal{C}(ask_1(t)$ fails without temporal transition, i.e. at time 1. This contradicts the phased embedding property since the agent $\mathcal{C}(ask_1(t))$ has to fail at time 2 after one temporal transition.

The proof of the second relation also proceeds by contradiction by reasoning on $nask_1(t)$. Using arguments similar to those exposed above, it is possible to prove that the normal form of $\mathcal{C}(nask_1(t))$ does not contain choices starting with a tell primitive and, consequently, is of the following form:

$$(ask_{d_1}(t_1)\ ;\ B_1) + \cdots + (ask_{d_m}(t_m)\ ;\ B_m)$$

for some durations $d_i$ and tokens $t_i$ with $m > 0$. It follows that any computation of $\mathcal{C}(nask_1(t))$ on the empty store fails or starts with a temporal transition. This contradicts the phased embedding property, since $\mathcal{C}(nask_1(t))$ has to succeed at time 1 without any temporal transition. □

The primitives $\{get, tell\}$ are strictly more expressive than the pair of primitives $\{ask, tell\}$. Moreover adding *ask* to $\mathcal{R}(get, tell)$ does not yield an additional expressiveness.

**Proposition 11.**

(i) $\mathcal{R}(ask, tell) \leq \mathcal{R}(get, tell)$
(ii) $\mathcal{R}(get, tell) \not\leq \mathcal{R}(ask, tell)$
(iii) $\mathcal{R}(get, tell) \equiv \mathcal{R}(ask, get, tell)$.

**Proof.** (i) Because of the infinite enumerability of the tokens, we associate with each token $t$ a pair of tokens that, for simplicity, we denote as $t_f$ and $t_i$. Intuitively, they correspond to a token $t$ on the store with a finite or infinite duration, respectively. As there are no *nask* primitives, temporal transitions decreasing the duration of finite tokens will occur only in the case of failing computations. In this context, there will be temporal transitions until the current store $\sigma$ satisfies $\sigma^- = \sigma$, i.e. until all tokens with finite duration disappear.

We can then define the coder $\mathcal{C}$ as follows, with $d_1, d_2 > 0$ and with $d_1$ finite:

$$\begin{aligned}\mathcal{C}(tell_0(t)) &= tell_0(t) \\ \mathcal{C}(tell_{d_1}(t)) &= tell_{d_1}(t)\ ;\ tell_\infty(t_f) \\ \mathcal{C}(tell_\infty(t)) &= tell_\infty(t_i) \\ \mathcal{C}(ask_0(t)) &= (get_0(t_f)\ ;\ tell_\infty(t_f)) + (get_0(t_i)\ ;\ tell_\infty(t_i)) \\ \mathcal{C}(ask_{d_2}(t)) &= (get_{d_2}(t_f)\ ;\ tell_\infty(t_f)) + (get_{d_2}(t_i)\ ;\ tell_\infty(t_i)).\end{aligned}$$

The associated decoder $\mathcal{D}^e$ is defined by :

$$\mathcal{D}^e((\sigma,\delta)) = \begin{cases}(\sigma_\infty,\delta^-) & \text{if } \delta=\delta^-\\ (\sigma_f,\delta^+) & \text{if } \delta=\delta^+\end{cases}$$

where $\sigma_\infty = \{t : t_{i_\infty} \in \sigma\}$ and $\sigma_f = \{t : \exists d > 0 : t_d \in \sigma\}$.

(ii) Assume that $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, tell)$ and consider $tell_1(a)\ ;\ get_1(a)$. Since $\mathcal{C}$ is compositional and since $\mathcal{O}_r(tell_1(a)\ ;\ get_1(a)) = \{(\emptyset,\delta^+)\}$, the termination mark of any element of $\mathcal{O}_r(\mathcal{C}(tell_1(a))\ ;\ \mathcal{C}(get_1(a)))$ is successful. As $\mathcal{C}(get_1(a))$ is composed of *ask* and *tell* primitives only and since *ask*, *tell* primitives do not destroy elements, it follows that any element of $\mathcal{O}_r(\mathcal{C}(tell_1(a))\ ;\ \mathcal{C}(get_1(a))\ ;\ \mathcal{C}(get_1(a)))$ has a successful termination mark. However, $\mathcal{O}_r(tell_1(a)\ ;\ get_1(a)\ ;\ get_1(a)) = \{(\emptyset,\delta^-)\}$ which contradicts property $P_3$.

(iii) The inequality $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, get, tell)$ follows directly from the inclusion of languages. To prove the converse inequality, we consider the coder of point (i), extended by $\mathcal{C}(get_0(t)) = get_0(t_f) + get_0(t_i)$ and $\mathcal{C}(get_{d_2}(t)) = get_{d_2}(t_f) + get_{d_2}(t_i)$. □

The languages $\mathcal{R}(ask, nask, tell)$ and $\mathcal{R}(get, tell)$ are incomparable. To establish this property, we introduce an auxiliary lemma.

**Lemma 2.** *For any agent $A$ in $\mathcal{R}(ask, nask, tell)$, if $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \tau\rangle$ then for some $\tau' \subseteq \tau$: $\langle A \,||\, A \mid \sigma\rangle \mapsto^* \langle B \,||\, B \mid \tau \cup \tau'\rangle$ where $\cup$ denotes union on multisets.*

**Proof.** The proof is conducted by induction on the number of steps of the computation $\langle A \mid \sigma\rangle \mapsto^* \langle B \mid \tau\rangle$. □

**Proposition 12.** *$\mathcal{R}(get, tell) \not\leq \mathcal{R}(ask, nask, tell)$ and $\mathcal{R}(nask, tell) \not\leq \mathcal{R}(get, tell)$.*

**Proof.** (i) Let us proceed by contradiction and assume that $\mathcal{R}(get, tell) \leq \mathcal{R}(ask, nask, tell)$. In that case, as $\mathcal{O}_r(tell_1(t)\ ;\ get_1(t)) = \{(\emptyset,\delta^+)\}$, any computation of $A = \mathcal{C}(tell_1(t))\ ;\ \mathcal{C}(get_1(t))$ starting in the empty store is successful by $P_3$. By Lemma 2, there is a computation of $B = \mathcal{C}(tell_1(t))\ ;\ (\mathcal{C}(get_1(t)) \,||\, \mathcal{C}(get_1(t)))$ starting in the empty store that is successful, which contradicts, by $P_2$ and $P_3$, the fact that $\mathcal{O}_r(tell_1(t)\ ;\ (get_1(t) \,||\, get_1(t))) = \{(\emptyset,\delta^-)\}$.

(ii) The proof is similar to that of Proposition 10(ii). □

The hierarchy study is completed by observing the following fact: if the language contains the *nask* and *tell* primitives, their association with *get* is not sufficient to express the *ask* primitive. This is formulated in the following proposition.

**Proposition 13.** *One has*

(i) $\mathcal{R}(ask, nask, tell) \not\leq_s \mathcal{R}(nask, get, tell)$
(ii) $\mathcal{R}(ask, nask, get, tell) \not\leq_s \mathcal{R}(nask, get, tell)$.

**Proof.** We prove the first relation. The second one is deduced from it and from Proposition 8 by transitivity of $\leq_s$.

Let us proceed by contradiction. Assume that $\mathcal{R}(ask, nask, tell) \leq_s \mathcal{R}(nask, get, tell)$ and consider a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ provided by Definition 31.

Consider $\mathcal{C}(ask_\infty(t))$. Since it is in $\mathcal{R}(nask, get, tell)$, its normal form can be written as

$$\begin{aligned}\mathcal{C}(ask_\infty(t)) = {} & (get_{d_1}(t_1)\ ;\ A_1) + \cdots + (get_{d_n}(t_n)\ ;\ A_n)\\ & + (nask_{e_1}(u_1)\ ;\ B_1) + \cdots + (nask_{e_m}(u_m)\ ;\ B_m)\\ & + (tell_{f_1}(s_1)\ ;\ C_1) + \cdots + (tell_{f_l}(s_l)\ ;\ C_l)\end{aligned}$$

for some tokens $t_i$, $u_i$ and $s_i$, with $n, m, l \geq 0$.

Moreover, one may assume without lost of generality that there is no choice starting with a $nask_0$ primitive. Indeed, if this was the case, this primitive will never compute and the behavior of the coding will be completely the same as the behavior of the same agent without this choice.

A second observation is that $\mathcal{C}(ask_\infty(t))$ cannot contain any choice starting with a $nask_e$ with $e > 0$ or a *tell* primitive. Indeed, if the first alternative starting with a $nask_e$ with $e > 0$ primitive exists, the agent $\mathcal{C}(ask_\infty(t) + tell_\infty(t)))$ accepts the following prefix of derivation:

$$\langle \mathcal{C}(ask_\infty(t) + tell_\infty(t)) \mid \emptyset \rangle_1 \rightarrow \langle B_1 \mid \sigma \rangle_1.$$

As $ask_\infty(t)$ fails, this derivation provides a valid prefix for a failing computation. This contradicts, by property $P_3$, the fact that the agent $ask_\infty(t) + tell_\infty(t)$ has only successful computations starting at time 1 on the empty store.

The absence of choice starting with a *tell* primitive can be shown similarly.

Consequently, the agent $\mathcal{C}(ask_\infty(t))$ has a normal form of the following type:

$$\mathcal{C}(ask_\infty(t)) = (get_{d_1}(t_1) \,;\, A_1) + \cdots + (get_{d_n}(t_n) \,;\, A_n). \tag{1}$$

The second part of the proof consists in building an agent including $ask_\infty(t)$ and showing that its coding falsifies property $P_3$.

Let $M$ denote the biggest finite duration of the *tell* primitives occurring in the $A_i$'s.

Let us now consider the agent $tell_{M+1}(t)$. By the phase-simulation property, its coder admits a computation without temporal steps of the following type:

$$\langle \mathcal{C}(tell_{M+1}(t)) \mid \emptyset \rangle_1 \rightarrow^* \langle E \mid \sigma t \rangle_1$$

for some store $\sigma t$. Let us denote by $\tau$ the sub-multiset composed of the $t_i$'s included in $\sigma t$ and $K$ the number of tokens of this multiset.

Let us now consider a computation of an agent that will take all those $t_i$'s out of the store. To that end, we take the coder of the agent

$$AG = tell_{M+1}(t) \,;\, (||_{i=1}^{K} ask_\infty(t)) \tag{2}$$

where $||_{i=1}^{K} ask_\infty(t)$ denotes $K$ parallel execution of $ask_\infty(t)$. Its coder admits the computation consisting of the execution of the coded version of the *tell* primitive and then, for each parallel *ask* primitive, one execution of a $get_{d_j}(t_j)$ primitive. Formally, for some store $\sigma$,

$$\begin{aligned} \langle \mathcal{C}(tell_{M+1}(t) \,;\, ||_{i=1}^{K} ask_\infty(t)) \mid \emptyset \rangle_1 &\rightarrow^* \langle \mathcal{C}(||_{i=1}^{K} ask_\infty(t)) \mid \sigma t \rangle_1 \\ &\rightarrow^* \langle ||_{i=1}^{K} A_{j(i)} \mid \sigma t \setminus \tau \rangle_1 \\ &\rightarrow^* \langle E \mid \sigma \rangle_1 \end{aligned} \tag{3}$$

where $j(i)$ denotes the alternative selected in the $i^{\text{th}}$ parallel execution of $\mathcal{C}(ask_\infty(t))$.

The store $\sigma t \setminus \tau$ contains no token $t_i$. Moreover, the $t_i$'s occurring in $\sigma$ follow from the execution of one of the $A_i$ agents.

The following two reasoning steps examine the behavior of this agent $AG$ and the content of $\sigma$. The third one concludes by providing an agent falsifying property $P_3$.

*Step 1. There is at least one $t_i$ in $\sigma$.*

Proceed ab absurdo and consider the agent

$$tell_{M+1}(t) \,;\, (||_{i=1}^{K} ask_\infty(t)) \,;\, ask_\infty(t).$$

If $\sigma$ contains none of the $t_i$'s, the above computation of $\mathcal{C}(tell_{M+1}(t) \,;\, ||_{i=1}^{K} ask_\infty(t))$ provides a valid prefix for a failing computation of its coder. By property $P_3$, this contradicts the fact that this agent accepts only successful computations.

*Step 2. At least one of the $t_i$'s occurring in $\sigma$ has an infinite duration.*

Proceed again by contradiction. Following the choice of $M$, the $t_i$'s with finite duration expire at the latest at time $M + 1$.

Now, we consider an agent that forces temporal steps. This agent is

$$AM1 = tell_M(u) \,;\, nask_{M+1}(u)$$

for some token $u$ different from $t$. This agent and the agent $AG$ defined in (2) admit a parallel computation without interaction. Indeed,

$$\begin{aligned} \langle AG \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \{t_{[1:M+1]}\}\rangle_1 \\ \langle tell_M(u)\ ;\ nask_{M+1}(u) \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \emptyset\rangle_{M+1} \\ \langle AG \mid\mid AM1 \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \{t_1\}\rangle_{M+1}. \end{aligned}$$

By the phase-simulable property, the coder of $tell_M(u)\ ;\ nask_{M+1}(u)$ accepts the following computation

$$\begin{aligned} \langle \mathcal{C}(tell_M(u)\ ;\ nask_{M+1}(u)) \mid \emptyset\rangle_1 &\to \langle nask_{M+1}(u) \mid \sigma u\rangle_1 \\ &\rightsquigarrow^M \langle \mathcal{C}(nask_{M+1}(u)) \mid \sigma u'\rangle_{M+1} \\ &\to^* \langle E \mid \tau\rangle_{M+1} \end{aligned} \tag{4}$$

for some stores $\sigma u$, $\sigma u'$ and $\tau$. Moreover, none of the $get_{d_i}(t_i)$ primitives occurring in (1) can compute on $\tau$ at time $M+1$. Indeed if that was the case, this should be a valid prefix for a failing computation of the coder of $tell_M(u)\ ;\ nask_{M+1}(u)\ ;\ (ask_\infty(t)\ +\ ask_\infty(u))$. That contradicts by property $P_3$ the fact that this agent admits only successful computations.

Given derivation (4) of $\mathcal{C}(AM1)$ and (3) of $\mathcal{C}(AG)$, property $P_4$ ensures the existence of a computation of their parallel composition of the following type.

$$\begin{aligned} &\langle \mathcal{C}((tell_M(u)\ ;\ nask_{M+1}(u))) \mid\mid (\mathcal{C}(tell_{M+1}(t)\ ;\ (||_{i=1}^K ask_\infty(t)))) \mid \emptyset\rangle_1 \\ &\quad\mapsto^* \langle E \mid \sigma^{-M} \cup \tau\rangle_{M+1} \end{aligned}$$

where $\sigma^{-M}$ denotes the store $\sigma$ after $M$ temporal steps. This is a valid prefix for a failing computation of the coder of

$$(AM1 \mid\mid AG)\ ;\ (ask_\infty(t)) \tag{5}$$

which, by $P_3$, contradicts the fact that the agent accepts only successful computations.

*Step 3. Contradiction*

We have now the information that there is one $(t_i)_\infty$ in $\sigma$. Moreover, the reasoning on the derivation of the coding of the agent (5) of step 2 may be used to conclude that at least one of the corresponding $get_{d_i}(t_i)$ of the coding (1) has a positive duration $d_i$. Let $I$ denote one of them.

Like in step 2, we consider the agent

$$AM2 = tell_{M+1}(v)\ ;\ nask_{M+2}(v)$$

for some token $v$ different from $t$ and the following computation of its coder:

$$\begin{aligned} \langle \mathcal{C}(tell_{M+1}(v)\ ;\ nask_{M+2}(v)) \mid \emptyset\rangle_1 &\to \langle nask_{M+2}(v)) \mid \sigma v\rangle_1 \\ &\rightsquigarrow^{M+1} \langle \mathcal{C}(nask_{M+2}(v) \mid \sigma v'\rangle_{M+2} \\ &\to^* \langle E \mid \tau\rangle_{M+2} \end{aligned}$$

for some stores $\sigma v$, $\sigma v'$ and $\tau$. Moreover none of the $get_{d_i}(t_i)$ primitives occurring in (1) can compute on $\tau$ at time $M+2$.

Like in step 2, we observe that $tell_{M+1}(v)\ ;\ nask_{M+2}(v)$ and $AG$ defined by (2) admit a parallel computation without interaction. Property $P_4$ ensures the existence of a computation the following type.

$$\begin{aligned} &\langle \mathcal{C}(tell_{M+1}(v)\ ;\ nask_{M+2}(v)) \mid\mid (\mathcal{C}(tell_{M+1}(t)\ ;\ (||_{i=1}^K ask_\infty(t)))) \mid \emptyset\rangle_1 \\ &\quad\mapsto^* \langle E \mid \sigma^{-(M+1)} \cup \tau\rangle_{M+2} \end{aligned}$$

where $\sigma^{-(M+1)}$ denotes the store $\sigma$ after $M+1$ temporal steps.

The coder of the agent

$$(AM2 \mid\mid AG)\ ;\ ask_\infty(t)$$

has only failing computations. Moreover, it accepts the following computation prefix:

$$\begin{aligned}&\langle \mathcal{C}((AM2 \mid\mid AG)\ ;\ ask_{\infty}(t)) \mid \emptyset\rangle_1\\ &\quad \mapsto^* \langle \mathcal{C}(ask_{[0:\infty]}(t)) \mid \sigma^{-(M+1)} \cup \sigma v\rangle_{M+2}\\ &\quad \rightarrow \langle A_I \mid \sigma^{-(M+1)} \cup \sigma v\rangle_{M+2}\end{aligned}$$

where $\sigma^{-(M+1)}$ denote the store $\sigma$ after $M+1$ temporal steps. This provides also a valid prefix for a failing computation of the coder of the following agent

$$(AM2 \mid\mid AG)\ ;\ (ask_{[0:\infty]}(t)\ +\ tell_{[0:\infty]}(t)).$$

By property $P_3$, this contradicts the fact that this agent has only successful computations. □

### 4.3. The hierarchy of the languages with wait

The wait primitive being of the same nature as the delay primitive, it turns out that the results and the proofs of Section 4.1 can be transposed to the $\mathcal{W}$ family of languages. Consequently, we shall just mention the results in this section and refer the interested reader to [36] for details of the proofs.

**Proposition 14.**

(1) $\mathcal{W}(X) \leq \mathcal{W}(Y)$, *for all subsets* $X$ *and* $Y$ *of* $\{ask, nask, get, tell\}$ *such that* $X \subseteq Y$.
(2) $\mathcal{W}(ask, tell) \not\leq \mathcal{W}(tell)$ *and* $\mathcal{W}(nask, tell) \not\leq \mathcal{W}(tell)$.
(3) $\mathcal{W}(ask, tell) \not\leq \mathcal{W}(nask, tell)$ *and* $\mathcal{W}(nask, tell) \not\leq \mathcal{W}(ask, tell)$.
(4) $\mathcal{W}(ask, tell) \leq \mathcal{W}(get, tell)$ *and* $\mathcal{W}(get, tell) \not\leq \mathcal{W}(ask, tell)$.
(5) $\mathcal{W}(get, tell) \equiv \mathcal{W}(ask, get, tell)$.
(6) $\mathcal{W}(nask, get, tell) \equiv \mathcal{W}(ask, nask, get, tell)$.
(7) $\mathcal{W}(nask, tell) \not\leq \mathcal{W}(get, tell)$ *and* $\mathcal{W}(get, tell) \not\leq \mathcal{W}(ask, nask, tell)$.

### 4.4. The hierarchy of the languages with time intervals

Immediate results for languages with time intervals are that sublanguages are embedded in super-languages and that the empty language is strictly less powerful than any non-empty languages.

**Proposition 15.** *For all subsets* $X$ *and* $Y$ *of* $\{ask, nask, get, tell\}$ *such that* $X \subseteq Y$, *one has* $\mathcal{I}(X) \leq \mathcal{I}(Y)$.

In the $\mathcal{I}(tell, X)$ family of languages the embedding relation is equivalent to the inclusion one. This is shown by the three following propositions.

**Proposition 16.** $\mathcal{I}(nask, tell) \not\leq \mathcal{I}(ask, get, tell)$.

**Proof.** Let us proceed by contradiction. Assume that $\mathcal{I}(nask) \leq \mathcal{I}(ask, get, tell)$ and consider a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ satisfying $P_1$ to $P_3$ and the phase-simulation property.

Observe the coding of $nask_{[1:1]}(t)$. Since it is in $\mathcal{I}(ask, get, tell)$, it can be considered equivalent to the following normal form:

$$\begin{aligned}\mathcal{C}(nask_{[1:1]}(t)) = {} & (ask_{[b_1:e_1]}(t_1)\ ;\ A_1)\ +\ \cdots\ +\ (ask_{[b_n:e_n]}(t_n)\ ;\ A_n)\\ & +\ (get_{[c_1:f_1]}(u_1)\ ;\ B_1)\ +\ \cdots\ +\ (get_{[c_m:f_m]}(t_m)\ ;\ B_m)\\ & +\ (tell_{[d_1:g_1]}(s_1)\ ;\ C_1)\ +\ \cdots\ +\ (tell_{[d_l:g_l]}(s_l)\ ;\ C_l)\end{aligned}$$

for some tokens $t_i$, $u_i$ and $s_i$, with $n, m, l \geq 0$.

By property $P_3$ any execution of the coder on the empty set at time 1 succeeds. Consequently, $l \geq 1$, namely there is at least one alternative with a tell primitive. The proof is then concluded by establishing that the $d_i$'s must be strictly greater than 1 but have to be strictly smaller than 2, which is absurd.

Indeed, consider $d_1$, the proof being similar for other $d_i$'s. Assume $d_1 \leq 1$. In this case, by the phase-simulation property, the coding of $nask_{[2:2]}(t)\ ;\ nask_{[1:1]}(t)$ accepts a prefix of a failing computation of the following type

$$\begin{aligned}&\langle \mathcal{C}(nask_{[2:2]}(t))\ ;\ \mathcal{C}(nask_{[1:1]}(t)) \mid \emptyset\rangle_1\\ &\mapsto^*\ \langle \mathcal{C}(nask_{[1:1]}(t)) \mid \sigma\rangle_2\\ &\to\ \langle C_1 \mid \sigma'\rangle_2\end{aligned}$$

for some stores $\sigma$ and $\sigma'$. This is also a valid prefix for a failing computation of the coder of $nask_{[2:2]}(t)\ ;\ (nask_{[1:1]}(t) + nask_{[2:2]}(t))$. By property $P_3$, this contradicts the fact that this agent has only successful computations.

Assume now that any $d_i$ is greater than 2. In this case, any computation of the coder begins as follows

$$\langle \mathcal{C}(nask_{[1:1]}(t)) \mid \emptyset\rangle_1 \rightsquigarrow \langle \mathcal{C}(nask_{[1:1]}(t)) \mid \emptyset\rangle_2.$$

By the phase-simulation property, this contradicts the fact that $nask_{[1:1]}(t)$ succeeds without any temporal step. □

**Proposition 17.** $\mathcal{I}(ask, tell) \not\leq_s \mathcal{I}(nask, get, tell)$.

**Proof.** The proof consists of an adaptation of the proof of Proposition 13. Let us thus proceed by contradiction and assume that $\mathcal{I}(ask, tell) \leq_s \mathcal{I}(nask, get, tell)$ by means of a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ characterized by Definition 31.

Following arguments similar to those of the proof of Proposition 13, it may be proved that $\mathcal{C}(ask_{[0:\infty]}(t))$ has a normal form of the following type:

$$\mathcal{C}(ask_{[0:\infty]}(t)) = (get_{[b_1:e_1]}(t_1)\ ;\ A_1) + \cdots + (get_{[b_n:e_n]}(t_n)\ ;\ A_n) \tag{6}$$

Let $M$ denote the biggest finite time upper bound of the *tell* primitives occurring in the $A_i$'s.

In these conditions, the rest of the proof consists in establishing that $\mathcal{C}(ask_{[0:\infty]}(t))$ has to contain some $get_{[b_{i(j)}:M+j]}(t_{i(j)})$ primitive for any $j$. As the coding has to be finite this provides the announced contradiction.

*Case* $j = 1$

Let us now consider the agent $tell_{[0:M+1]}(t)$. By the phase-simulable property, its coder admits a computation without temporal steps of the following type:

$$\langle \mathcal{C}(tell_{[0:M+1]}(t) \mid \emptyset\rangle_1 \to^* \langle E \mid \sigma t\rangle_1$$

for some store $\sigma t$. Let us denote by $\tau$ the submultiset composed of the $t_i$'s included in $\sigma t$ and $K$ the number of tokens of this multiset.

Let us now consider a computation of an agent that will take all those $t_i$'s out of the store. To that end, we take the coder of the agent

$$AG = tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^{K} ask_{[0:\infty]}(t)) \tag{7}$$

where $||_{i=1}^{K} ask_{[0:\infty]}(t)$ denotes $K$ parallel execution of $ask_{[0:\infty]}(t)$. Its coder admits the computation consisting of the execution of the coded version of the *tell* primitive and then, for each parallel *ask* primitive, one execution of a $get_{[b_j:e_j]}(t_j)$ primitive. Formally, for some store $\sigma$,

$$\begin{aligned}\langle \mathcal{C}(tell_{[0:M+1]}(t)\ ;\ \ ||_{i=1}^{K} ask_{[0:\infty]}(t)) \mid \emptyset\rangle_1 &\to^* \langle \mathcal{C}(||_{i=1}^{K} ask_{[0:\infty]}(t)) \mid \sigma t\rangle_1\\ &\to^* \langle ||_{i=1}^{K} A_{j(i)} \mid \sigma t \setminus \tau\rangle_1\\ &\to^* \langle E \mid \sigma\rangle_1\end{aligned} \tag{8}$$

where $j(i)$ denotes the alternative selected in the $i$th parallel execution of $\mathcal{C}(ask_{[0:\infty]}(t))$.

The store $\sigma t \setminus \tau$ contains no token $t_i$. Moreover, the $t_i$'s occurring in $\sigma$ follow from the execution of one of the $A_i$ agents.

The following three reasoning steps examine the behavior of this agent $AG$ and conclude that the coding (6) has to contain some $get_{[b_i:M+1]}(t_i)$ primitive.

*Step 1. There is at least one* $t_i$ *in* $\sigma$.

This is established in a similar way to Proposition 13 by considering the agent

$$tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^{K} ask_{[0:\infty]}(t))\ ;\ ask_{[0:\infty]}(t).$$

*Step 2. At least one of the $t_i$'s occurring in $\sigma$ has an infinite duration.*

Proceed again by contradiction. Following the choice of $M$, these $t_i$'s expire at the latest at time $M$.

Now, we consider an agent that forces temporal steps. This agent is $tell_{[M+1:\infty]}(u)$ for some token $u$ different from $t$. This agent and $AG$ defined in (7) admit a parallel computation without interaction. Indeed,

$$\begin{aligned}\langle AG \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \{t_{[1:M+1]}\}\rangle_1\\ \langle tell_{[M+1:\infty]}(u) \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \{u_{[M+1:\infty]}\}\rangle_{M+1}\\ \langle AG \mid\mid tell_{[M+1:\infty]}(u) \mid \emptyset\rangle_1 &\mapsto^* \langle E \mid \{t_{[M+1:M+1]}, u_{[M+1:\infty]}\}\rangle_{M+1}.\end{aligned}$$

By the phase-simulable property, the coder of $tell_{[M+1:\infty]}(u)$ accepts the following computation

$$\begin{aligned}\langle \mathcal{C}(tell_{[M+1:\infty]}(u)) \mid \emptyset\rangle_1 &\rightsquigarrow^M \langle \mathcal{C}(tell_{[M+1:\infty]}(u)) \mid \emptyset\rangle_{M+1}\\ &\to^* \langle E \mid \sigma u\rangle_{M+1}\end{aligned} \tag{9}$$

for some store $\sigma u$. Moreover, none of the $get_{[b_i:e_i]}(t_i)$ primitives occurring in (6) can compute on $\sigma u$ at time $M+1$. Indeed if that was the case, this should be a valid prefix for a failing computation of the coder of $tell_{[M+1:\infty]}(u)\ ;\ (ask_{[0:\infty]}(t) + ask_{[0:\infty]}(u))$. That contradicts by property $P_3$ the fact that this agent admits only successful computations.

Given derivation (9) of $\mathcal{C}(tell_{[M+1:\infty]}(u))$ and (8) of $\mathcal{C}(AG)$, property $P_4$ ensures the existence of a computation of their parallel composition of the following type.

$$\begin{aligned}&\langle \mathcal{C}(tell_{[M+1:\infty]}(u)) \mid\mid (\mathcal{C}(tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^K ask_{[0:\infty]}(t)))) \mid \emptyset\rangle_1\\ &\quad\mapsto^* \langle E \mid \sigma^{-M} \cup \sigma u\rangle_{M+1}\end{aligned}$$

where $\sigma^{-M}$ denotes the store $\sigma$ after $M$ temporal steps. This is a valid prefix for a failing computation of the coder of

$$[(tell_{[M+1:\infty]}(u)) \mid\mid (tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^K ask_{[0:\infty]}(t)))]\ ;\ (ask_{[0:\infty]}(t)) \tag{10}$$

which, by $P_3$, contradicts the fact that the agent accepts only successful computations.

*Step 3. Coding (6) has to contain some $get_{[b_i:M+1]}(t_i)$ primitive.*

We have now the information that there is one $(t_i)_{[1:\infty]}$ in $\sigma$. Let $N$ denote the biggest of the $e_i$ upper bounds of the $get_{[b_i:e_i]}(t_i)$ occurring the coding (6) such that $(t_i)_{[1:\infty]}$ is in $\sigma$.

The proof of the third step is obtained by showing that $N = M+1$.

Firstly, assume, by contradiction, that $N < M+1$. The reasoning led on the derivation of the coding of the agent (10) of step 2 may be against used to conclude. Indeed none of the $get_{[b_i:e_i]}(t_i)$ can compute on the store $\sigma^{-M} \cup \sigma u$ at time $M+1$.

Secondly, assume, again by contradiction, that $N < M+2$. Like in step 2, we consider the agent $tell_{[M+2:\infty]}(v)$ for some token $v$ different from $t$ and the following computation of its coder:

$$\langle \mathcal{C}(tell_{[M+2:\infty]}(v) \mid \emptyset\rangle_1 \rightsquigarrow^{M+1} \langle \mathcal{C}(tell_{[M+2:\infty]}(v) \mid \emptyset\rangle_{M+2} \to^* \langle E \mid \sigma v\rangle_{M+2}$$

for some store $\sigma v$ on which none of the $get_{[b_i:e_i]}(t_i)$ primitives occurring in (6) can compute at time $M+2$.

Like in step 2, we observe that $tell_{[M+2:\infty]}(v)$ and $AG$ defined by 7 admit a parallel computation without interaction. Property $P_4$ ensures the existence of a computation the following type.

$$\begin{aligned}&\langle \mathcal{C}(tell_{[M+2:\infty]}(u)) \mid\mid (\mathcal{C}(tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^K ask_{[0:\infty]}(t)))) \mid \emptyset\rangle_1\\ &\qquad\mapsto^* \langle E \mid \sigma^{-(M+1)} \cup \sigma u\rangle_{M+2}\end{aligned}$$

where $\sigma^{-(M+1)}$ denotes the store $\sigma$ after $M+1$ temporal steps.

The coder of the agent

$$[(tell_{[M+2:\infty]}(v)) \mid\mid (tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^K ask_{[0:\infty]}(t)))]\ ;\ ask_{[0:\infty]}(t)$$

has only failing computations. Moreover, it accepts the following computation prefix:

$$\begin{aligned}&\langle [(tell_{[M+2:\infty]}(v)) \mid\mid (tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^K ask_{[0:\infty](t)}))]\ ;\ ask_{[0:\infty]}(t) \mid \emptyset\rangle_1\\ &\qquad\mapsto^* \langle \mathcal{C}(ask_{[0:\infty]}(t)) \mid \sigma^{-(M+1)} \cup \sigma v\rangle_{M+2}\\ &\qquad\to \langle A_N \mid \sigma^{-(M+1)} \cup \sigma v\rangle_{M+2}\end{aligned}$$

where $\sigma^{-(M+1)}$ denotes the store $\sigma$ after $M+1$ temporal steps. This provides also a valid prefix for a failing computation of the coder of the following agent

$$[(tell_{[M+2:\infty]}(v)) \;||\; (tell_{[0:M+1]}(t)\;;\;(||_{i=1}^{K} ask_{[0:\infty]}(t)))]\;;\;(ask_{[0:\infty]}(t)\;+\;tell_{[0:\infty]}(t)).$$

By property $P_3$, this contradicts the fact that this agent has only successful computations.

The two previous contradictions force us to conclude that $N = M+1$. Following the definition of $N$ this leads to the conclusion that coding (6) has to contain some $get_{[b_i:M+1]}(t_i)$ primitive.

*General case :* $j > 1$

The reasoning concerning $AG = tell_{[0:M+1]}(t)\;;\;(||_{i=1}^{K} ask_{[0:\infty]}(t))$ can be followed as regards the similarly defined agents

$$AG_j \;=\; tell_{[0:M+j]}(t)\;;\;(||_{i=1}^{K(j)} ask_{[0:\infty]}(t))$$

for any $j$. This leads to the conclusion that the coding (6) has to contain some $get_{[b_{i(j)}:M+j]}(t_{i(j)})$ primitive for any $j$. As the coding has to be finite this provides the announced contradiction. □

A direct consequence of Proposition 17, given by the transitivity of the $\leq_s$ relation and the inclusion $\mathcal{I}(get, tell) \leq_s \mathcal{I}(nask, get, tell)$, is that $\mathcal{I}(ask, tell) \not\leq_s \mathcal{I}(get, tell)$. However, a strongest proposition can be established for the $\leq$ relationship.

**Proposition 18.** $\mathcal{I}(ask, tell) \not\leq \mathcal{I}(get, tell)$.

**Proof.** Assuming a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ provided by Definition 28 (instead of Definition 31), the proof basically follows the reasoning used for the previous proposition. Differences occur in steps 2 and 3 which have to be established without the hypothesis of parallel computations without interaction (property $P_4$). Using the notation introduced in the proof of Proposition 17, let us present the pieces of proof that have to be substituted for step 2 and step 3 in order to obtain a complete proof for our claim.

*Step 2′. At least one of the $t_i$'s occurring in $\sigma$ has an infinite duration.*

We proceed by contradiction. Following the choice of $M$, these $t_i$'s expire at least at time $M$. Now, we consider an agent that forces temporal steps. This agent is $tell_{[M+1:\infty]}(u)$ for some token $u$ different from $t$. By the phase-simulable property, its coder accepts the following computation

$$\begin{aligned}\langle \mathcal{C}(tell_{[M+1:\infty]}(u)) \mid \emptyset\rangle_1 &\rightsquigarrow^M \langle \mathcal{C}(tell_{[M+1:\infty]}(u)) \mid \emptyset\rangle_{M+1} \\ &\rightarrow^* \langle E \mid \sigma u\rangle_{M+1}\end{aligned} \tag{11}$$

for some store $\sigma u$. Moreover, none of the $get_{[b_i:e_i]}(t_i)$ primitives occurring in (6) can compute on $\sigma u$ at time $M+1$. Indeed if that was the case, this should be a valid prefix for a failing computation of the coder of $tell_{[M+1:\infty]}(u)\;;\;(ask_{[0:\infty]}(t)\;+\;ask_{[0:\infty]}(u))$. That contradicts by property $P_3$ the fact that this agent admits only successful computations.

One has also to observe that, as the coding is only composed of *get* and *tell* primitives, whatever the store is at time $M+1$, $\mathcal{C}(tell_{[M+1:\infty]}(u))$ admits a computation that ends at time $M+1$ and results in adding the tokens of $\sigma u$ on the store.

Let us now consider this agent in a sequential composition starting with $AG$ and finishing with $ask_{[0:\infty]}(t)$:

$$tell_{[0:M+1]}(t)\;;\;(||_{i=1}^{K} ask_{[0:\infty]}(t))\;;\;tell_{[M+1:\infty]}(u)\;;\;ask_{[0:\infty]}(t).$$

That sequentially composed agent accepts only successful computations. Moreover, its coder accepts the following computation

$$\begin{aligned}&\langle \mathcal{C}(tell_{[0:M+1]}(t)\;;\;(||_{i=1}^{K} ask_{[0:\infty]}(t))\;;\;tell_{[M+1:\infty]}(u)\;;\;ask_{[0:\infty]}(t)) \mid \emptyset\rangle_1 \\ &\quad\rightarrow^* \langle \mathcal{C}(tell_{[M+1:\infty]}(u)\;;\;ask_{[0:\infty]}(t)) \mid \sigma\rangle_1 \\ &\quad\rightsquigarrow^M \langle \mathcal{C}(tell_{[M+1:\infty]}(u)\;;\;ask_{[0:\infty]}(t)) \mid \sigma^{-M}\rangle_{M+1} \\ &\quad\rightarrow^* \langle \mathcal{C}(ask_{[0:\infty]}(t)) \mid \sigma^{-M} \cup \sigma u\rangle_{M+1}\end{aligned}$$

where $\sigma^{-M}$ denotes the store $\sigma$ after $M$ temporal steps. This is a valid prefix for a failing computation of the coder, which, by $P_3$, contradicts the fact that the agent accepts only successful computations.

*Step 3′. Coding (6) has to contain some $get_{[b_i:M+1]}(t_i)$ primitive.*

We have now the information that there is one $(t_i)_{[1:\infty]}$ in $\sigma$. Let $N$ denote the biggest of the $e_i$ upper bounds of the $get_{[b_i:e_i]}(t_i)$ occurring in the coding (6) such that $(t_i)_{[1:\infty]}$ is in $\sigma$.

The proof of the third step is obtained by showing that $N = M + 1$.

Firstly, assume, by contradiction, that $N < M + 1$. The reasoning followed on the derivation of the coding of the agent (10) of step $2'$ may be used again to conclude. Indeed none of the $get_{[b_i:e_i]}(t_i)$ can compute on the store $\sigma^{-M} \cup \sigma u$ at time $M + 1$.

Secondly, assume, again by contradiction, that $N < M + 2$. Like in step $2'$, we consider the agent $tell_{[M+2:\infty]}(v)$ for some token $v$ different from $t$ and the following computation of its coder:

$$\langle \mathcal{C}(tell_{[M+2:\infty]}(v) \mid \emptyset\rangle_1 \rightsquigarrow^{M+1} \langle \mathcal{C}(tell_{[M+2:\infty]}(v) \mid \emptyset\rangle_{M+2} \rightarrow^* \langle E \mid \sigma v\rangle_{M+2}$$

for some store $\sigma v$ on which none of the $get_{[b_i:e_i]}(t_i)$ primitives occurring in (6) can compute at time $M + 2$.

Like in step $2'$, we observe that $tell_{[M+2:\infty]}(v)$ is only composed of *get* and *tell* primitives and whatever the store is at time $M + 2$, $\mathcal{C}(tell_{[M+2:\infty]}(v))$ admits a computation that ends at time $M + 2$ and results in adding the tokens of $\sigma v$ on the store.

Now, the coder of the agent

$$(tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^{K} ask_{[0:\infty]}(t)))\ ;\ (tell_{[M+2:\infty]}(v))\ ;\ ask_{[0:\infty]}(t)$$

has only failing computations. Moreover, it accepts the following computation prefix:

$$\begin{aligned}
&\langle \mathcal{C}(tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^{K} ask_{[0:\infty]}(t))\ ;\ tell_{[M+2:\infty]}(u)\ ;\ ask_{[0:\infty]}(t)) \mid \emptyset\rangle_1 \\
&\quad \rightarrow^* \langle \mathcal{C}(tell_{[M+2:\infty]}(u)\ ;\ ask_{[0:\infty]}(t)) \mid \sigma\rangle_1 \\
&\quad \rightsquigarrow^{M+1} \langle \mathcal{C}(tell_{[M+2:\infty]}(u)\ ;\ ask_{[0:\infty]}(t)) \mid \sigma^{-(M+1)}\rangle_{M+2} \\
&\quad \rightarrow^* \langle \mathcal{C}(ask_{[0:\infty]}(t)) \mid \sigma^{-(M+1)} \cup \sigma u\rangle_{M+2}
\end{aligned}$$

where $\sigma^{-(M+1)}$ denotes the store $\sigma$ after $M + 1$ temporal steps. This provides also a valid prefix for a failing computation of the coder of the following agent

$$(tell_{[0:M+1]}(t)\ ;\ (||_{i=1}^{K} ask_{[0:\infty]}(t)))\ ;\ (tell_{[M+2:\infty]}(v))\ ;\ (ask_{[0:\infty]}(t)\ +\ tell_{[0:\infty]}(t)).$$

By property $P_3$, this contradicts the fact that this agent has only successful computations.

The two previous contradictions force us to conclude that $N = M + 1$. Following the definition of $N$ this leads to the conclusion that coding (6) has to contain some $get_{[b_i:M+1]}(t_i)$ primitive.

Given these modifications of steps 2 and 3, the proof concludes as the proof of Proposition 17. □

**Lemma 3.** *For any agent $A$ in $\mathcal{I}(ask, nask, tell)$, if $\langle A \mid \sigma\rangle_u \mapsto^* \langle B \mid \tau\rangle_v$ then for some $\tau' \subseteq \tau$:*

$$\langle A \mid\mid A \mid \sigma\rangle_u \mapsto^* \langle B \mid\mid B \mid \tau \cup \tau'\rangle_v$$

*where $\cup$ denotes union on multisets.*

**Proof.** The proof is obtained by induction on the number of steps of the computation $\langle A \mid \sigma\rangle_u \mapsto^* \langle B \mid \tau\rangle_v$. □

**Proposition 19.** $\mathcal{I}(get, tell) \not\leq \mathcal{I}(ask, nask, tell)$.

**Proof.** Let us proceed by contradiction and assume that $\mathcal{I}(get, tell) \leq \mathcal{I}(ask, nask, tell)$. In that case, as $\mathcal{O}_i(tell_{[1:1]}(t)\ ;\ get_1(t)) = \{(\emptyset, \delta^+)\}$, any computation of $A = \mathcal{C}(tell_{[1:1]}(t))\ ;\ \mathcal{C}(get_{[1:1]}(t))$ starting in the empty store is successful by $P_3$. Lemma 3 gives that, as a consequence, there is a computation of $B = \mathcal{C}(tell_{[1:1]}(t))\ ;\ (\mathcal{C}(get_{[1:1]}(t))\ ||\ \mathcal{C}(get_{[1:1]}(t)))$ starting in the empty store that is successful, which contradicts, by $P_2$ and $P_3$, the fact that $\mathcal{O}_i(tell_{[1:1]}(t)\ ;\ (get_{[1:1]}(t)\ ||\ get_{[1:1]}(t))) = \{(\emptyset, \delta^-)\}$. □

## 5. Inter-family comparisons

Let us now turn to the comparison of languages of different families. This is subsequently achieved by first comparing $\mathcal{L}$ with the relative time families $\mathcal{D}$ and $\mathcal{R}$, by then comparing $\mathcal{L}$ and the absolute time families $\mathcal{W}$ and $\mathcal{I}$, and finally by comparing the most expressive languages of the five families.

### 5.1. Comparing the $\mathcal{L}$, $\mathcal{D}$, and $\mathcal{R}$ families

#### 5.1.1. Comparing the $\mathcal{L}$ and $\mathcal{D}$ families

The comparison between the $\mathcal{L}$ and $\mathcal{D}$ families is substantiated by the results presented in this section together with those established in [34] and in Section 4.1.

The main observation here is that the $\mathcal{D}(X)$ language is strictly more expressive than the corresponding $\mathcal{L}(X)$ and no $\mathcal{L}$ is more expressive than a $\mathcal{D}$ language. In other words, the *delay* primitive cannot be expressed by (any combination of) the other primitives.

The first result is that, as intuitively expected, for the same set of primitives $X$, the language $\mathcal{D}(X)$ is more powerful than $\mathcal{L}(X)$.

**Proposition 20.** *For any* $X \subseteq \{ask, nask, get, tell\}$, $\mathcal{L}(X) \preceq \mathcal{D}(X)$.

The *delay* primitive cannot be expressed in any $\mathcal{L}(X)$.

**Proposition 21.** *For any* $X, Y \subseteq \{ask, nask, get, tell\}$, *then* $\mathcal{D}(X) \not\preceq \mathcal{L}(Y)$.

**Proof.** A coding of the primitive *delay*(2) satisfying the phased embedding property has to terminate successfully at time 2. However, any agent of $\mathcal{L}(Y)$ ends at time 1. □

In addition to the $\mathcal{L}(X) \preceq \mathcal{D}(X)$ inclusions of property 20, one has the quite unexpected following inequality.

**Proposition 22.** $\mathcal{L}(ask, nask) \preceq \mathcal{D}(nask, tell)$.

**Proof.** Indeed the primitives of $\mathcal{L}(ask, nask)$ are unable to update the initial empty store. Therefore any *ask* primitive always fails whereas any *nask* primitive always succeeds. This may be modelled by taking the coder $\mathcal{C}$ defined as follows: for any $t \in Stoken$

$$\mathcal{C}(ask(t)) = tell(t)\ ;\ nask(t)$$
$$\mathcal{C}(nask(t)) = tell(t).$$

The decoder $\mathcal{D}^e$ to be considered then maps any store to the empty store while conserving termination marks. □

In the rest of the section we show that there are no other inclusions between those two hierarchies. This corresponds to the fact that the *delay* primitive is not able to express any other primitive.

**Proposition 23.**
(i) $\mathcal{L}(ask, tell) \not\preceq \mathcal{D}(nask, tell)$
(ii) $\mathcal{L}(tell, nask) \not\preceq \mathcal{D}(ask, get, tell)$.

**Proof.** Similar to that of Proposition 4. □

#### 5.1.2. Comparing the $\mathcal{L}$ and $\mathcal{R}$ families

The first result in the comparison between the $\mathcal{L}(X)$ and $\mathcal{R}(X)$ families is that, as intuitively expected, for the same set of primitives $X$, the language $\mathcal{R}(X)$ is more expressive than the language $\mathcal{L}(X)$.

**Proposition 24.** $\mathcal{L}(X) \preceq \mathcal{R}(X)$, *for any* $X \subseteq \{ask, nask, get, tell\}$.

The equivalence is obtained for the tell primitive.

**Proposition 25.** $\mathcal{L}(tell) \equiv \mathcal{R}(tell)$.

**Proof.** The inclusion $\mathcal{L}(tell) \preceq \mathcal{R}(tell)$ is immediate. As any agent of $\mathcal{R}(tell)$ succeeds at time 1, the converse is obtained by considering the coder $\mathcal{C}$ defined by $\mathcal{C}(tell_d(t)) = tell(t)$ and the identity as decoder. □

Associated with any other primitive, *tell* distinguishes the two families $\mathcal{L}$ and $\mathcal{R}$.

**Proposition 26.** $\mathcal{R}(tell, X) \not\preceq \mathcal{L}(Y)$, *for any nonempty* $X \subseteq \{ask, nask, get\}$ *and any* $Y \subseteq \{ask, nask, get, tell\}$.

**Proof.** The proof consists of providing in each of the $\mathcal{R}(tell, X)$ languages an agent whose computation contains at least one temporal transition. Indeed, as no agent of $\mathcal{L}(Y)$ computes with a temporal transition, it is then not possible to find in $\mathcal{L}(Y)$ a coding of this agent satisfying the phased embedding property.

The agents to be considered are as follows:

- if $ask \in X$ then consider $ask_2(t)$,
- if $nask \in X$ then consider $tell_1(t)\ ;\ nask_2(t)$,
- if $get \in X$ then consider $get_2(t)$,
- if $X$ contains a combination of one of the *ask*, *nask*, *get* primitives then consider one of the corresponding agents. □

*5.1.3. Comparing the $\mathcal{D}$ and $\mathcal{R}$ families*

We now compare the $\mathcal{D}$ and $\mathcal{R}$ families. The first main observation is that the *delay* primitive cannot be expressed in any $\mathcal{R}(X)$ language.

**Proposition 27.** *For any $X, Y \subseteq \{ask, nask, get, tell\}$, one has $\mathcal{D}(X) \not\leq \mathcal{R}(Y)$.*

**Proof.** By contradiction, suppose that $\mathcal{D}(X) \leq \mathcal{R}(Y)$ and consider a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ which satisfy properties $P_1$ to $P_3$ and the phase-simulation property.

By the phase-simulation property, the coding of the agent $delay(0)$ succeeds at time 1.

We now consider the agent $delay(1)$. By the phase-simulation property, the coding of $delay(1)$ succeeds at time 2. The first step of any such computation corresponds to the execution of a $tell_d(t)$ or $nask_d(t)$ primitive on the empty set and thus is not a temporal step. Any computation can then be represented as follows.

$$\langle \mathcal{C}(delay(1)) \mid \emptyset\rangle_1 \to \langle C' \mid \sigma\rangle_1 \to^* \langle C'' \mid \tau\rangle_1 \rightsquigarrow \langle \mathcal{C}(delay(0)) \mid \tau^-\rangle_2 \to^* \langle E \mid \mu\rangle_2$$

where $\mathcal{D}^e((\tau, \delta^+)) = (\emptyset, \delta^+)$ and $\mathcal{D}^e((\mu, \delta^+)) = (\emptyset, \delta^+)$.

As the first step is not a temporal transition, this gives, by definition of $+$, a valid prefix for a computation of the coding of the agent $delay(0) + delay(1)$ that finishes at time 2. That contradicts the fact that, by the phase-simulation property, any computation of this agent succeeds at time 1. □

There is thus no embedding of a language of the $\mathcal{D}$ family in the $\mathcal{R}$ family. Consider now the possibility of converse relations.

As a trivial consequence of Propositions 20 and 25, one has the following property.

**Proposition 28.** $\mathcal{R}(tell) \leq \mathcal{D}(tell)$.

A $\mathcal{R}$ language containing the tell primitive associated with any other primitive cannot be expressed in the corresponding language of the $\mathcal{D}$ family.

**Proposition 29.** $\mathcal{R}(nask, tell) \not\leq \mathcal{D}(ask, nask, get, tell)$.

**Proof.** Since $\mathcal{D}(ask, nask, get, tell) \equiv \mathcal{D}(nask, get, tell)$, it is sufficient to prove that $\mathcal{R}(nask, tell) \not\leq \mathcal{D}(nask, get, tell)$.

By contradiction, suppose that $\mathcal{R}(nask, tell) \leq \mathcal{D}(nask, get, tell)$ and consider a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ satisfying properties $P_1$ to $P_3$. The proof is based on the examination of the normal form of the coding of the primitives *nask*.

For any $i \in Stime$, the agent $\mathcal{C}(nask_i(t))$ is in $\mathcal{D}(ask, nask, get, tell)$, and its normal form can then be written as

$$\begin{aligned}\mathcal{C}(nask_i(t)) = {} & (delay(j_1)\ ;\ A_1) + \cdots + (delay(j_n)\ ;\ A_n) \\ & + (nask(t_1)\ ;\ B_1) + \cdots + (nask(t_m)\ ;\ B_m) \\ & + (get(u_1)\ ;\ C_1) + \cdots + (get(u_l)\ ;\ C_l) \\ & + (tell(v_1)\ ;\ D_1) + \cdots + (tell(v_k)\ ;\ D_k)\end{aligned}$$

where $n, m, l, k \geq 0$.

According to the phase-simulation property, denote by $\sigma t_j$ ($j \in Stime$) any store such that

$$\langle \mathcal{C}(tell_j(t)) \mid \emptyset \rangle_1 \rightarrow^* \langle E \mid \sigma t_j \rangle_1.$$

Our first observation is that the coding cannot contain any choice starting with a *tell* or a *delay*(0) primitive, i.e. $k = 0$ and $j_x > 0$ ($1 \leq x \leq n$). Indeed, if there is one choice starting with a *tell* primitive, then the coding of the agent $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ accepts the following derivation

$$\begin{aligned}&\langle \mathcal{C}(tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))) \mid \emptyset \rangle_1\\ &\quad \rightarrow^* \langle \mathcal{C}(nask_i(t)\ +\ nask_1(s)) \mid \sigma t_{i+1} \rangle_1 \rightarrow \langle D_1 \mid \sigma t_{i+1} \cup \{v_1\} \rangle_1.\end{aligned}$$

As the computation of $tell_{i+1}(t)\ ;\ nask_i$ fails, this derivation provides a valid prefix for a failing derivation of the agent. That contradicts, by property $P_3$, the fact that $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ has only successful computations on the empty store. The absence of an alternative in the choice starting with a *delay*(0) primitive can be shown similarly.

The second observation is that any of the $nask(t_k)$ ($k = 1, \ldots, m$) and $get(u_k)$ ($k = 1, \ldots, l$) primitives appearing in the coding of $nask(i)$ fails on any $\sigma t_j$ ($j \in Stime$). Indeed if $nask(t_K)$ succeeds with $\sigma t_J$, then the coding of the agent $tell_J(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ has the following derivation

$$\begin{aligned}\langle \mathcal{C}(tell_J(t)\ ;\ (nask_i(t)\ +\ nask_1(s))) \mid \emptyset \rangle_1 &\rightarrow^* \langle \mathcal{C}(nask_i(t)\ +\ nask_1(s)) \mid \sigma t_J \rangle_1\\ &\rightarrow \langle B_K \mid \sigma t_J \rangle_1.\end{aligned}$$

On the one hand, if $J \leq i$ then $tell_J(t)\ ;\ nask_i(t)$ fails and this provides a valid prefix for a failing derivation of the agent. On the other hand, if $J > i$, this derivation provides a successful derivation with a final configuration decoded as $(\delta^+, \emptyset)$. Both cases contradict, by property $P_3$, the fact that the semantics of $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_1(s))$ is $\{(\delta^+, \{v\})\}$. The absence of successful $get(u_k)$ on $\sigma t_j$ can be shown similarly.

The third observation is about the *delay* primitives appearing in the coding. None of the $j_1, \ldots, j_l > 0$ can have 1 as value. Indeed, if $j_J = 1$, in view of our second observation, the coding of the agent $tell_i(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t))$ accepts the following derivation

$$\begin{aligned}&\langle \mathcal{C}(tell_i(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t)) \mid \emptyset \rangle_1\\ &\quad \rightarrow^* \langle \mathcal{C}(nask_i(t)\ +\ nask_{i+1}(v)) \mid \sigma t_i \rangle_1\\ &\quad \rightsquigarrow \langle \ldots\ +\ (delay(0)\ ;\ A_J)\ +\ \ldots \mid \sigma t_i{}^- \rangle_2\\ &\quad \rightarrow \langle A_J \mid \sigma t_i{}^- \rangle_2.\end{aligned}$$

As $tell_i(t)\ ;\ nask_i(t)$ fails, this derivation provides a valid prefix for a failing derivation of the agent. That contradicts, by property $P_3$, the fact that $tell_{i+1}(t)\ ;\ (nask_i(t)\ +\ nask_{i+1}(t))$ has only successful computations on the empty store.

An inductive reasoning leads similarly to the property that no value of *Stime* is possible for $t_j$.

All these observations together lead to the fact that the coding of a $nask_i(t)$ primitive has a normal form of the following type:

$$\begin{aligned}\mathcal{C}(nask_i(t)) = &\ (nask(t_1)\ ;\ B_1)\ +\ \cdots\ +\ (nask(t_m)\ ;\ B_m)\\ &+ (get(u_1)\ ;\ C_1)\ +\ \cdots\ +\ (get(u_l)\ ;\ C_l)\end{aligned}$$

where every $nask(t_k)$ ($k = 1, \ldots, m$) and $get(u_k)$ ($k = 1, \ldots, l$) primitive appearing in the coding of $nask(i)$ fails on any $\sigma t_j$ ($j \in Stime$). Consequently, $\langle \mathcal{C}(tell_1(t)\ ;\ nask_2(t)) \mid \emptyset \rangle_1 \rightarrow^* \langle \mathcal{C}(nask_2(t)) \mid \sigma t_1 \rangle_1$ is a valid prefix for a failing computation of $\mathcal{C}(tell_1(t)\ ;\ nask_2(t))$. However, this contradicts, by property $P_3$, the fact that $tell_1(t)\ ;\ nask_2(t)$ has only successful computations. □

**Proposition 30.**

(i) $\mathcal{R}(ask, tell) \not\leq \mathcal{D}(ask, nask, get, tell)$

(ii) $\mathcal{R}(get, tell) \not\leq \mathcal{D}(ask, nask, get, tell)$.

**Proof.** (i) By contradiction, suppose that $\mathcal{R}(ask, tell) \leq \mathcal{D}(ask, nask, get, tell)$ and consider a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$ satisfying properties $P_1$ to $P_3$ and the phase-simulation property. The proof is based on the examination of the normal form of the coding of the $ask_1(t)$ and $ask_2(t)$ primitives.

*A. Coding of* $ask_1(t)$. As $\mathcal{C}(ask_1(t))$ is in $\mathcal{D}(ask, nask, get, tell)$, its normal form can then be written as

$$\begin{aligned}\mathcal{C}(ask_1(t)) = {} & (delay(d_1)\ ;\ A_1) + \cdots + (delay(d_j)\ ;\ A_j)\\ & + (ask(t_1)\ ;\ B_1) + \cdots + (ask(t_k)\ ;\ B_k)\\ & + (nask(u_1)\ ;\ C_1) + \cdots + (nask(u_l)\ ;\ C_l)\\ & + (get(v_1)\ ;\ D_1) + \cdots + (get(v_m)\ ;\ D_m)\\ & + (tell(w_1)\ ;\ E_1) + \cdots + (tell(w_n)\ ;\ E_n)\end{aligned}$$

for some times $d_i$ and tokens $t_i$, $u_i$, $v_i$ and $w_i$ where $j, k, l, m, n \geq 0$.

Our first observation is that the coding cannot contain any choice starting with a *tell*, a *nask* or a *delay*(0) primitive. Indeed, if there is one choice starting with a *tell* primitive, then the coding of the agent $tell_1(t) + ask_1(t)$ accepts the following derivation

$$\langle \mathcal{C}(tell_1(t) + ask_1(t)) \mid \emptyset\rangle_1 \to \langle E_1 \mid \{w_1\}\rangle_1.$$

As the computation of $ask_1(t)$ fails on the empty store this provides a valid prefix for a failing computation of the agent. That contradicts, by property $P_3$, the fact that $tell_1(t) + ask_1(t)$ has only successful computations. The absence of choice starting with a *nask* or a *delay*(0) primitive can be shown similarly.

The second observation concerns the choices of $\mathcal{C}(ask(t))$ starting with a $delay(d_i)$ primitive. There is at least one choice starting with a *delay* primitive. Moreover, at least one of the durations $d_i$ has to be 1. Indeed, on the one hand, if there is no choice starting with a *delay* primitive, as *ask* and *get* primitives fail on the empty store, the coding of $ask_1(t)$ fails on the empty store at time 1, without temporal transitions. On the other hand, if the smallest duration is greater than 1, the coding of $ask_1(t)$ accepts the following derivation

$$\langle \mathcal{C}(ask_1(t)) \mid \emptyset\rangle_1 \rightsquigarrow \langle \mathcal{C}(ask_1(t))^- \mid \emptyset\rangle_2 \rightsquigarrow \langle \mathcal{C}(ask_1(t))^{--} \mid \emptyset\rangle_3.$$

As the only computation of $ask_1(t)$ is $\langle ask_1(t) \mid \emptyset\rangle_1 \rightsquigarrow \langle ask_0(t) \mid \emptyset\rangle_2 \not\mapsto$ these two cases contradict the phase-simulation property.

With these first two observations, the normal form of the coding of $ask_1(t)$ can be written as follows.

As $\mathcal{C}(ask_1(t))$ is in $\mathcal{D}(ask, nask, get, tell)$, and its normal form can then be written as

$$\begin{aligned}\mathcal{C}(ask_1(t)) = {} & (delay(1)\ ;\ A_1) + (delay(d_2)\ ;\ A_2)\ldots + (delay(d_j)\ ;\ A_j)\\ & + (ask(t_1)\ ;\ B_1) + \cdots + (ask(t_k)\ ;\ B_k)\\ & + (get(v_1)\ ;\ D_1) + \cdots + (get(v_m)\ ;\ D_m)\end{aligned}$$

for some times $d_i$ and tokens $t_i$ and $v_i$ where $j, k, m \geq 0$ and $d_i \geq 1$ $(2 \leq i \leq j)$. As any *ask* and *get* primitive fails on the empty store, by the phase-simulation property, this coding admits the following computation.

$$\langle \mathcal{C}(ask_1(t)) \mid \emptyset\rangle_1 \rightsquigarrow \langle delay(0)\ ;\ A_1 + \ldots \mid \emptyset\rangle_2 \to \langle A_1 \mid \emptyset\rangle_2 \to^* \langle A_1' \mid \sigma\rangle_2 \not\mapsto .$$

*B. Coding of* $ask_2(t)$. A reasoning similar to the one used for $ask_1(t)$ establishes that the coder of $ask_2(t)$ does not contain any choice starting with a *tell*, a *nask* or a *delay*(0) primitive.

*C. Conclusion.* The proof is concluded by observing that the coding of $ask_1(t) + ask_2(t)$ admits the following derivation:

$$\begin{aligned}\langle \mathcal{C}(ask_1(t) + ask_2(t)) \mid \emptyset\rangle_1 &\rightsquigarrow \langle delay(0)\ ;\ A_1 + \cdots + \mathcal{C}(ask_2(t))^- \mid \emptyset\rangle_2\\ &\to \langle A_1 \mid \emptyset\rangle_2 \to^* \langle A_1' \mid \sigma\rangle_2 \not\mapsto .\end{aligned}$$

By the phase-simulation property, this contradicts the fact that any computation of $ask_1(t) + ask_2(t)$ fails at time 3.

(ii) The proof of the inequality $\mathcal{R}(get, tell) \not\leq \mathcal{D}(ask, nask, get, tell)$ is obtained by using the same reasoning on $get_1(t) + get_2(t)$. □

*5.1.4. A relative timed language embedding the $\mathcal{D}$ and $\mathcal{R}$ families*

The results obtained in the previous subsections and summarized on Fig. 6 show that the relative timed families of languages admit two most expressive languages: $\mathcal{R}(ask, nask, get, tell)$ and $\mathcal{D}(ask, nask, get, tell)$. By Propositions 27 and 29, neither of them is able to embed the other one. This suggests the idea of introducing a "super"-language that embeds those two leading languages. Proposition 27 shows that the inability of $\mathcal{R}(ask, nask, get, tell)$ to embed $\mathcal{D}(ask, nask, get, tell)$ comes from the fact that it fails to express the *delay* primitive. This observation leads us to consider the language obtained by adding the *delay* primitive and its corresponding $D2$ transition rule to $\mathcal{R}(ask, nask, get, tell)$. We denote the resulting language by $\mathcal{S}$.

Let us now prove that $\mathcal{S}$ embeds the languages $\mathcal{R}(ask, nask, get, tell)$ and $\mathcal{D}(ask, nask, get, tell)$ in a phased modular way.

**Proposition 31.**
(i) $\mathcal{R}(ask, nask, get, tell) < \mathcal{S}$
(ii) $\mathcal{D}(ask, nask, get, tell) < \mathcal{S}$.

**Proof.** Let us first consider the two embedding relations. The absence of converse embeddings will then directly follow from the transitivity of the $\leq$ relationship.

(i) $\mathcal{R}(ask, nask, get, tell) < \mathcal{S}$ is obviously given by considering the identity as coder and decoder functions.

(ii) $\mathcal{D}(ask, nask, get, tell) < \mathcal{S}$ is obtained by considering the coder $\mathcal{C}$ defined as follows.

$$\begin{aligned} &\mathcal{C}(ask(t)) = ask_\infty(t) && \mathcal{C}(get(t)) = get_\infty(t) \\ &\mathcal{C}(nask(t)) = nask_\infty(t) && \mathcal{C}(tell(t)) = tell_\infty(t). \\ &\mathcal{C}(delay(d)) = delay(d) \end{aligned}$$

Moreover the decoder $\mathcal{D}^e$ is defined by $\mathcal{D}^e((\sigma, \delta)) = (\sigma^*, \delta)$ where $\sigma^*$ denotes the multisets of the tokens occurring in $\sigma$ without their subscripted duration. □

*5.2. Comparing the $\mathcal{L}$, $\mathcal{W}$, and $\mathcal{I}$ families*

Let us now turn to the absolute time families $\mathcal{W}$ and $\mathcal{I}$.

A first observation is that introducing absolute time is a safe and necessary extension to the Linda family. Rephrased in more formal terms, the $\mathcal{L}$ family of languages can be embedded in the $\mathcal{W}$ and $\mathcal{I}$ families. The converse properties do not hold.

**Proposition 32.** *For any $X, Y, Z \subseteq \{ask, nask, get, tell\}$ such that $X \neq \emptyset$,*
(i) $\mathcal{L}(Y) \leq \mathcal{W}(Y)$ *and* $\mathcal{W}(Y) \not\leq \mathcal{L}(Z)$
(ii) $\mathcal{L}(X) \leq \mathcal{I}(X)$ *and* $\mathcal{I}(X) \not\leq \mathcal{L}(Y)$.

**Proof.** (i) The modular phased embedding from $\mathcal{L}(Y)$ into $\mathcal{W}(Y)$ is given by considering the identity as coder $\mathcal{C}$ and decoder $\mathcal{D}^e$.

The absence of modular phased embedding from $\mathcal{W}(Y)$ into $\mathcal{L}(Z)$ follows from the fact that no agent of $\mathcal{L}(Z)$ is able to execute a temporal step. It is then impossible to find a coder and a decoder satisfying the phase-simulation property.

(ii) The modular phased embedding from $\mathcal{L}(X)$ into $\mathcal{I}(X)$ is given by considering the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ defined in the following way:

$$\begin{aligned} \mathcal{C}(ask(t)) &= ask_{[1:\infty]}(t) \\ \mathcal{C}(nask(t)) &= nask_{[1:\infty]}(t) \\ \mathcal{C}(get(t)) &= get_{[1:\infty]}(t) \\ \mathcal{C}(tell(t)) &= tell_{[1:\infty]}(t) \\ \mathcal{D}^e((\sigma, \delta)) &= (\sigma_\infty, \delta) \end{aligned}$$

where $\sigma_\infty = \{t : t_{[1:\infty]} \in \sigma\}$. These $\mathcal{C}$ and $\mathcal{D}^e$ obviously satisfy the properties $P_1$ to $P_3$. The phase-simulable property is a direct consequence of the fact that the computations of agents of $\mathcal{L}(X)$ contain no temporal transition.

The absence of modular phased embedding from $\mathcal{I}(X)$ into $\mathcal{L}(Y)$ follows from the fact that no agent of $\mathcal{L}(X)$ is able to execute a temporal step. It is then impossible to find a coder and a decoder satisfying the phase-simulation property. □

Let us now turn to the comparison between the $\mathcal{W}$ and $\mathcal{I}$ families of languages. An immediate result is that the *ask* and *get* primitives are not sufficient to express the *wait* primitive. In contrast, the *nask* and *tell* primitives are sufficient to express the *wait* primitive. Moreover, the *tell* primitive is not powerful enough to distinguish the two families.

**Proposition 33.** *$\mathcal{W}(X) \leq \mathcal{I}(X)$ for any $X \subseteq \{ask, nask, get, tell\}$ such that $tell \in X$.*

**Proof.** Thanks to the enumerability of the tokens, we can chose one unused token $tko$. The modular phased embedding from $\mathcal{W}(X)$ into $\mathcal{I}(X)$ is given by considering – the restriction to $X$ of – the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ defined in the following way:

$$\begin{aligned}
\mathcal{C}(wait(i)) &= tell_{[i:i]}(tko)\\
\mathcal{C}(ask(t)) &= ask_{[1:\infty]}(t)\\
\mathcal{C}(nask(t)) &= nask_{[1:\infty]}(t)\\
\mathcal{C}(get(t)) &= get_{[1:\infty]}(t)\\
\mathcal{C}(tell(t)) &= tell_{[1:\infty]}(t)\\
\mathcal{D}^e((\sigma, \delta)) &= (\sigma_\infty, \delta)
\end{aligned}$$

where $\sigma_\infty = \{t : t_{[1:\infty]} \in \sigma\}$. They obviously satisfy properties $P_1$ to $P_3$. The phase-simulation property is a direct consequence of the facts that the *wait* primitive is the only one leading to temporal transitions and that temporal transitions in $\mathcal{W}(X)$ do not modify the store. □

**Proposition 34.** *$\mathcal{I}(tell) \leq \mathcal{W}(tell)$.*

**Proof.** Thanks the enumerability of the set of the tokens, we associate with each time $i$ a token $tt_i$ and with each pair time $e$—token $t$ a token $t_e$. The modular phased embedding from $\mathcal{I}(tell)$ into $\mathcal{W}(tell)$ is given by considering the coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ defined in the following way:

$$\begin{aligned}
\mathcal{C}(tell(t_{[b:e]})) &= wait(b)\ ;\ tell(tt_i)\ ;\ tell(t_e)\\
\mathcal{D}^e((\sigma, \delta)) &= (\sigma_a, \delta)
\end{aligned}$$

where $\sigma_a = \{t : \exists e > I : t_e \in \sigma\}$ with $I$ the maximum of times such that $tt_i \in \sigma$. □

The combination of the *tell* primitive with any other one renders the $\mathcal{I}$ languages inexpressible in the $\mathcal{W}$ family.

**Proposition 35.**

(i) $\mathcal{I}(ask, tell) \not\leq \mathcal{W}(ask, nask, get, tell)$
(ii) $\mathcal{I}(get, tell) \not\leq \mathcal{W}(ask, nask, get, tell)$
(iii) $\mathcal{I}(nask, tell) \not\leq \mathcal{W}(ask, nask, get, tell)$.

**Proof.** (i) By contradiction, assume that $\mathcal{I}(ask, tell) \leq \mathcal{W}(ask, nask, get, tell)$. The proof is based on the examination of the behaviors of the agent $ask_{[2:2]}(t)$ and its coder. The only valid computation of this agent is the following failing computation.

$$\langle ask_{[2:2]}(t) \mid \emptyset\rangle_1 \rightsquigarrow \langle ask_{[2:2]}(t) \mid \emptyset\rangle_2 \not\mapsto .$$

By the phase-simulation property, any execution of its coder finishes at time 2.

The coding of the agent $ask_{[2:2]}(t)$ has a normal form of the following type:

$$\begin{aligned}\mathcal{C}(ask_{[2:2]}(t))) = {} & (wait(j_1)\ ;\ A_1) + \cdots + (wait(j_n)\ ;\ A_n)\\ & + (ask(t_1)\ ;\ B_1) + \cdots + (ask(t_m)\ ;\ B_m)\\ & + (nask(s_1)\ ;\ C_1) + \cdots + (nask(s_l)\ ;\ C_l)\\ & + (get(u_1)\ ;\ D_1) + \cdots + (get(u_o)\ ;\ D_o)\\ & + (tell(v_1)\ ;\ E_1) + \cdots + (tell(v_p)\ ;\ E_p)\end{aligned}$$

for some tokens $j_i, t_i, s_i, u_i$ and $v_i$, with $n, m, l, o, p \geq 0$.

Our first observation is that the coding cannot contain any choice starting with a *nask*, *tell*, *wait*(0) or *wait*(1) primitive. Indeed, if there was one starting with a *nask* primitive, by property $P_3$ and the phase-simulation property, the agent $\mathcal{C}(ask_{[2:2]}(t))$ would accept a derivation failing at time 2 of the following type

$$\langle \mathcal{C}(ask_{[2:2]}(t)) \mid \emptyset\rangle_1 \rightarrow \langle C_1 \mid \emptyset\rangle_1 \mapsto^* \langle C' \mid \sigma\rangle_2 \not\mapsto .$$

This is also a valid computation finishing at time 2 for the coder of the agent $ask_{[2:2]}(t) + ask_{[3:3]}(t)$. This contradicts the fact that, by the phase-simulation property, all the computations of this agent finish at time 3.

The absence of choice starting with a *tell*, a *wait*(0) or a *wait*(1) primitive can be shown similarly.

Consequently, the agent $\mathcal{C}(ask_{[2:2]})$ has a normal form of the following type

$$\begin{aligned}\mathcal{C}(ask_{[2:2]}(t))) = {} & (wait(j_1)\ ;\ A_1) + \cdots + (wait(j_n)\ ;\ A_n)\\ & + (ask(t_1)\ ;\ B_1) + \cdots + (ask(t_m)\ ;\ B_m)\\ & + (get(u_1)\ ;\ D_1) + \cdots + (get(u_o)\ ;\ D_o)\end{aligned}$$

for some tokens $j_i, t_i$ and $u_i$, with $j_i > 1$ and $n, m, o \geq 0$.

The second observation is that none of the $j_i$'s is 2. Indeed, an argument similar to the first observation ensures that the coder of $ask_{[3:3]}(t)$ has the same normal form as $\mathcal{C}(ask_{[2:2]}(t)))$. Assuming $j_I = 2$, the coder admits the following computation

$$\langle \mathcal{C}(ask_{[2:2]}(t)) \mid \emptyset\rangle_1 \rightsquigarrow \langle \mathcal{C}(ask_{[2:2]}(t)) \mid \emptyset\rangle_2 \rightarrow \langle A_I \mid \emptyset\rangle_2 \rightarrow^* \langle A' \mid \sigma\rangle_2 \nrightarrow .$$

This provides a valid computation for $\mathcal{C}(ask_{[2:2]}(t)) + \mathcal{C}(ask_{[3:3]}(t))$ finishing at time 2. This contradicts, by the phase-simulation property, the fact that the only computation of $ask_{[2:2]}(t) + ask_{[3:3]}(t)$ is

$$\begin{aligned}&\langle ask_{[2:2]}(t) + ask_{[3:3]}(t) \mid \emptyset\rangle_1 \rightsquigarrow \langle ask_{[2:2]}(t) + ask_{[3:3]}(t) \mid \emptyset\rangle_2\\ &\quad \rightsquigarrow \langle ask_{[2:2]}(t) + ask_{[3:3]}(t) \mid \emptyset\rangle_3 \not\mapsto .\end{aligned}$$

The final contradiction results from the following two facts. On the one hand, if the smallest of the $j_i$'s is greater than 3, computations of the coder finish at times greater than 3. On the other hand, if there is no $j_i$, i.e. if $n = 0$, computations fail at time 1. These two situations contradict, by the phase-simulation property, the fact that the only computation of $ask_{[2:2]}(t)$ finishes at time 2.

(ii) The inequality $\mathcal{I}(get, tell) \not\leq \mathcal{W}(ask, nask, get, tell)$ is obtained similarly to point (i) by examining $get_{[2:2]}(t)$.
(iii) The inequality $\mathcal{I}(nask, tell) \not\leq \mathcal{W}(ask, nask, get, tell)$ is obtained similarly by examining the agent $nask_{[2:2]}(t)$. □

### 5.3. Comparing relative time and absolute time languages

Sections 5.1 and 5.2 have presented the results needed to build, on the one hand, the expressiveness hierarchy of the relative time languages and, on the other hand, the expressiveness hierarchy of absolute time languages. The next natural question is to compare the relative time and absolute time languages.

Due to the very different natures of time they embody, the $\mathcal{R}$ and $\mathcal{I}$ families cannot be related in a compositional way. Nevertheless, a positive result is that it is possible to do so with the help of an auxiliary function returning the current time. This function is subsequently called $\nu$.

The time nature difference however forces the coders and decoders using the $\nu$ function to violate the compositionality property during time. As a result, languages extended with the $\nu$ function cannot be compared from the modular phased embedding point of view. We shall therefore adapt the embedding to just take into account

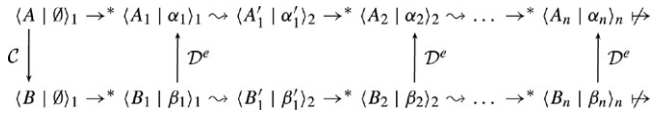


Fig. 5. Translation.

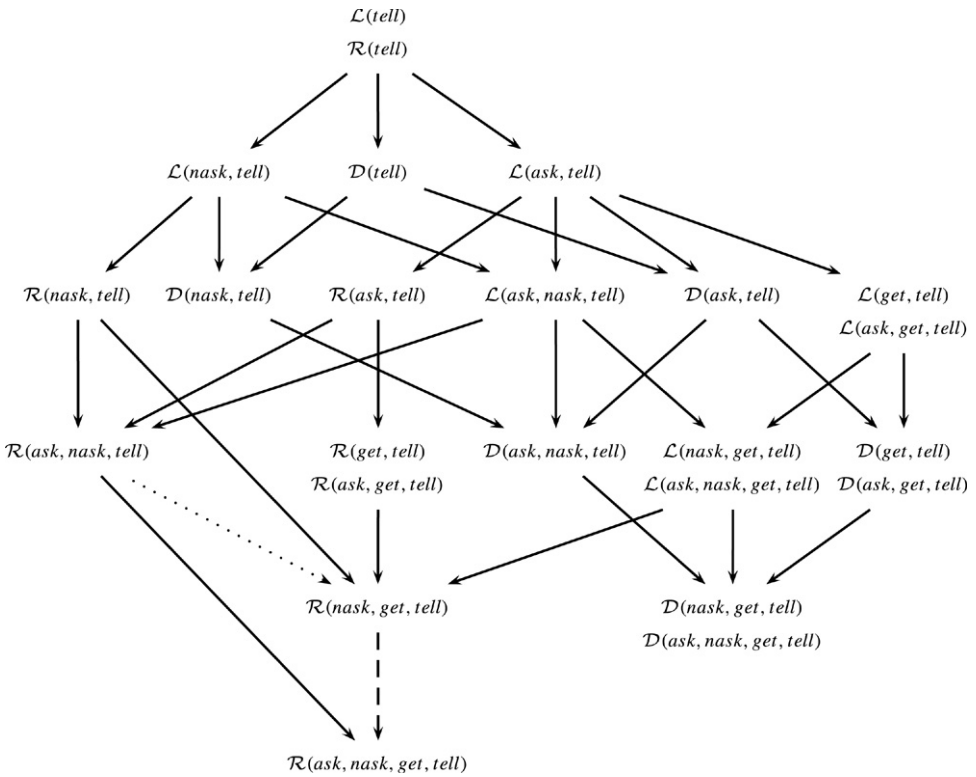


Fig. 6. Languages with relative time comparison.

elements of the behaviors detectable by an observer external to the agents. Fundamentally, the observable elements are the content of the store and the occurrences of temporal transitions.

On a point of notation, any language $L(X)$ extended with the $\nu$ function is subsequently denoted by $L_\nu(X)$. We will then declare that there is a translation from $L_\nu$ to $L'_\nu$ if there is a coder from $L_\nu$ in $L'_\nu$ such that there is a correspondence between the computations of any agent of $L_\nu$ and the computations of its coding in $L'_\nu$ from the point of view of temporal transitions and contents of the store. This is suggested by Fig. 5 and formalized in the following definition.

**Definition 32.** Let $L_\nu$ and $L'_\nu$ be two timed languages extended by the $\nu$ function. The coder $\mathcal{C}$ and decoder $\mathcal{D}^e$ provide a translation from $L_\nu$ into $L'_\nu$ iff for any agent $A$ of $L_\nu$ the following two properties hold.

(1) For any computation of $A$ of the following type

$$\langle A \mid \emptyset\rangle_1 \to^* \langle A_1 \mid \alpha_1\rangle_1 \rightsquigarrow \langle A'_1 \mid \alpha'_1\rangle_2 \to^* \langle A_2 \mid \alpha_2\rangle_2 \rightsquigarrow \ldots \to^* \langle A_n \mid \alpha_n\rangle_n \not\to$$

there is a computation of $B = \mathcal{C}(A)$ of the following type

$$\langle B \mid \emptyset\rangle_1 \to^* \langle B_1 \mid \beta_1\rangle_1 \rightsquigarrow \langle B'_1 \mid \beta'_1\rangle_2 \to^* \langle B_2 \mid \beta_2\rangle_2 \rightsquigarrow \ldots \to^* \langle B_n \mid \beta_n\rangle_n \not\to$$

such that $\mathcal{D}^e((\beta_i, \delta^-)) = (\alpha_i, \delta^-)$, for any $i < n$, and one of the following conditions is satisfied following $A_n$

(a) $A_n = E$ and $B_n = E$ and $\mathcal{D}^e((\beta_n, \delta^+)) = (\alpha_n, \delta^+)$.

(b) $A_n \neq E$ and $B_n \neq E$ and $\mathcal{D}^e((\beta_n, \delta^-)) = (\alpha_n, \delta^-)$.

(2) Conversely, for any computation of $B = \mathcal{C}(A)$ of the following type

$$\langle B \mid \emptyset\rangle_1 \rightarrow^* \langle B_1 \mid \beta_1\rangle_1 \rightsquigarrow \langle B'_1 \mid \beta'_1\rangle_2 \rightarrow^* \langle B_2 \mid \beta_2\rangle_2 \rightsquigarrow \ldots \rightarrow^* \langle B_n \mid \beta_n\rangle_n \not\mapsto$$

there is a computation of $A$ of the following type

$$\langle A \mid \emptyset\rangle_1 \rightarrow^* \langle A_1 \mid \alpha_1\rangle_1 \rightsquigarrow \langle A'_1 \mid \alpha'_1\rangle_2 \rightarrow^* \langle A_2 \mid \alpha_2\rangle_2 \rightsquigarrow \ldots \rightarrow^* \langle A_n \mid \alpha_n\rangle_n \not\mapsto$$

such that $\mathcal{D}^e((\beta_i, \delta^-)) = (\alpha_i, \delta^-)$, for any $i < n$, and one of the following conditions is satisfied following $B_n$

(a) $B_n = E$ and $A_n = E$ and $\mathcal{D}^e((\beta_n, \delta^+)) = (\alpha_n, \delta^+)$.

(b) $B_n \neq E$ and $A_n \neq E$ and $\mathcal{D}^e((\beta_n, \delta^-)) = (\alpha_n, \delta^-)$.

Given $L_\nu$ and $L'_\nu$ two timed languages extended by the $\nu$ function we will denote by $L_\nu \leq_t L'_\nu$ the fact that there is a translation from $L_\nu$ into $L'_\nu$.

In this subsection, we shall limit our investigations to the comparison of the most expressive languages. For the relative time languages, we consider the $\mathcal{S}$ language, and, for absolute time, the $\mathcal{I}(ask, nask, get, tell)$ language. Both of them are extended with the $\nu$ function, this resulting respectively in the languages $\mathcal{S}_\nu$ and $\mathcal{I}_\nu$. As shown below, the comparison is in favour of the absolute time languages: one has $\mathcal{S}_\nu \leq_t \mathcal{I}_\nu$. Actually, it is even possible to prove the following more general result.

**Proposition 36.** *For any $X \subseteq \{ask, nask, tell\}$, one has*

(i) $\mathcal{R}_\nu(X) \leq_t \mathcal{I}_\nu(X)$

(ii) $\mathcal{R}_\nu(X, delay) \leq_t \mathcal{I}_\nu(\{tell\} \cup X)$.

**Proof.** The two results are obtained by providing a coder $\mathcal{C}$ and a decoder $\mathcal{D}^e$, actually the same for the two results. Thanks to the denumerable property of the tokens, it is always possible to operate a token translation so as to identify two unused tokens $tko1$ and $tko2$. Given them, the coder $\mathcal{C}$ is defined as follows.

$$\begin{aligned}
\mathcal{C}(tell_0(t)) &= tell_{[0:0]}(t) \\
\mathcal{C}(tell_d(t)) &= tell_{[v:v+d-1]}(t) && \text{for any } d > 0 \\
\mathcal{C}(ask_d(t)) &= \textstyle\sum_{i=v}^{v+d-1} ask_{[i:i]}(t) \ + \ ask_{[v+d:v+d]}(tko1) && \text{for any finite } d \\
\mathcal{C}(ask_\infty(t)) &= ask_{[1:\infty]}(t) \\
\mathcal{C}(get_d(t)) &= \textstyle\sum_{i=v}^{v+d-1} get_{[i:i]}(t) \ + \ get_{[v+d:v+d]}(tko1) && \text{for any finite } d \\
\mathcal{C}(get_\infty(t)) &= get_{[1:\infty]}(t) \\
\mathcal{C}(nask_d(t)) &= \textstyle\sum_{i=v}^{v+d-1} nask_{[i:i]}(t) \ + \ ask_{[v+d:v+d]}(tko1) && \text{for any finite } d \\
\mathcal{C}(nask_\infty(t)) &= nask_{[1:\infty]}(t) \\
\mathcal{C}(delay(d)) &= tell_{[v+d:v+d]}(tko2) && \text{for any finite } d.
\end{aligned}$$

The decoder is defined by $\mathcal{D}^e((\sigma, \delta)) = (\sigma^*, \delta)$ where $\sigma^* = \{t_{e-b+1} : \exists b \leq e : t_{[b:e]} \in \sigma\}$. □

## 6. Implementation

In order to argue for the feasibility of the $\mathcal{D}$, $\mathcal{R}$, $\mathcal{W}$ and $\mathcal{I}$ families of languages, let us finally sketch how they can be implemented. This section actually reports on a prototype under development. It builds upon previous work carried out in order to implement the $\mathcal{L}$ family of languages (see, e.g., [13]).

### 6.1. Implementation of the Linda primitives

The implementation of the Linda primitives has been done by using the threads library of Solaris. The tuple space is implemented as a token-indexed list. Per list element (token), we keep track of the number of identical tokens (*token counter*), and of the input primitives that are suspended on this token. The token list is stored in shared memory. The list is directly updated by the communication primitives. In order to guarantee exclusive access, the individual list elements are protected with a lock, which means that operations on different tokens can execute in parallel. Only adding list elements to or removing list elements from the global list requires the complete tuple space to be locked. The following algorithms are employed for the primitives.

Performing a *nask* primitive first checks whether the associated token is known to the tuple space (already has an element in the list of tokens). If not, the tuple space is locked, and a list element for the token is created. If the token counter equals zero, the *nask* primitive succeeds. If not, it suspends, and is added to the list of suspended primitives, until the token counter reaches zero.

Asking a token $t$ first checks whether at least one occurrence of $t$ is present in the tuple space. If so, the primitive succeeds. Otherwise, the ask primitive is put in the associated list of waiting processes, until the token counter is positive.

Getting a token $t$ proceeds similarly but decrements the token counter for $t$. If the token counter reaches zero, we check whether there are suspended $nask(t)$ primitives. If so, the process associated with the *nask* primitive is resumed and is removed from the list of waiting primitives.

Finally, telling a token $t$ proceeds dually. The list of waiting processes is first inspected to discover an *ask* or *get* primitive waiting for $t$. In a case where an *ask* primitive is discovered, it is resumed and the search continues. In a case where a *get* primitive is discovered the token $t$ is consumed by that primitive and the corresponding process is resumed. If no waiting *get* primitives are encountered, then the token counter for $t$ is incremented.

As might be appreciated by the above description, the most distinguishing characteristic of our implementation is that the tuple space is completely passive. This approach offers considerable advantages over a process-based approach.

- It is more robust. Even after a crash, one can inspect the contents of the tuple space. This is more difficult if the contents is stored in a data structure that is passed around all the time.
- Communication with the tuple space does not need an intervention of a scheduler. For instance, getting a token is nothing more than changing some fields of a data structure. In the process-based approach, the process in charge of the tuple space must be activated before the communication can take place.
- In theory, the current tuple space can allow simultaneous updates of its substructures (one per token). A process version can only handle one request at a time (unless we create one process per token, which would require too many resources).

### 6.2. Implementing time

The implementation of the timed primitives is simplified by the observation that, thanks to the results of Section 5.3, it is sufficient to implement the primitives dealing with absolute time, provided that the kernel keeps track of the current time. Note that any operating system provides a means to deliver to its processes the current time as well as to awake suspended processes after a given interval.

Consequently, with respect to the untimed implementation, the basic adaptations are, on the one hand, to associate a period of validity with the tokens and processes of the waiting lists, and, on the other hand, to use the waking-up facilities provided by the operating system kernel to force *wait* primitives to succeed when the specified waiting time has been reached, to force timed *ask*, *nask*, and *get* primitives to fail when their period of validity is over, and to remove tokens whose period of validity is over.

### 6.3. Correctness issues

It is here worth stressing that such an implementation rests on the hypothesis that the number of operations actually performed per time unit is small enough or, restated in other terms, that the granularity of the time unit is big enough. Indeed, although the theoretical model of Section 2 allows an unlimited number of operations to take place at the same moment, in practice any operation takes some time, even if it is very small. If the amount of work to be done in one time unit exceeds the available time, the model must be adapted by introducing additional wait primitives which means that some operations are effectively delayed to the following time units.

## 7. Conclusion

The paper has presented four extensions of Linda in order to introduce time in coordination languages. All of them are based on the two-phase functioning approach to real-time systems already employed by languages such as Lustre [19] and Esterel [9].

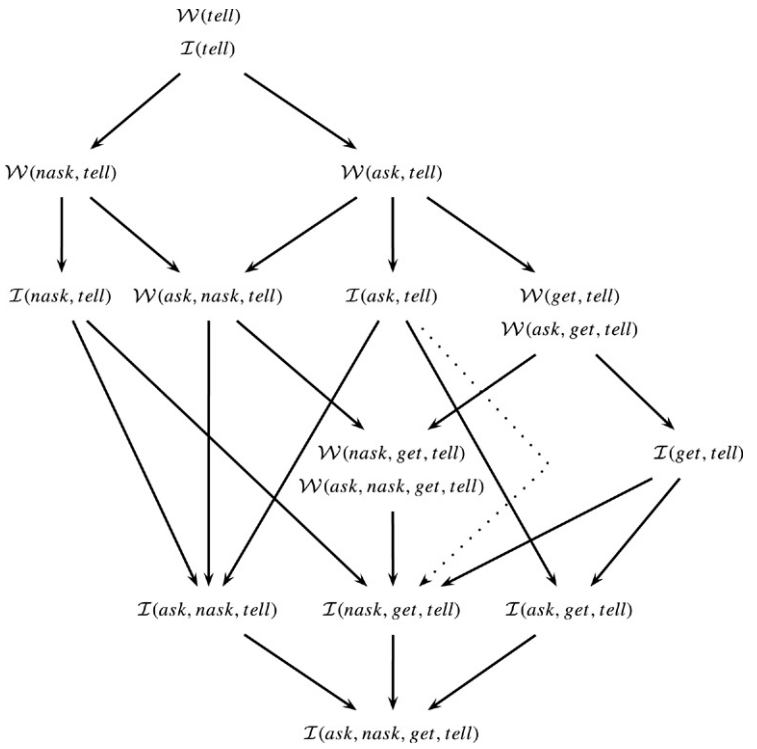


Fig. 7. Comparison of the $\mathcal{W}(X)$ and $\mathcal{I}(X)$ languages.

The resulting families of languages have been described by means of transition systems written in Plotkin's style. Their expressiveness has been studied by means of new concepts of embeddings, called modular phased embedding and strong modular phased embeddings. The complete expressiveness hierarchy of each family has been examined. We have also compared the expressiveness of languages of the different families. All these results are summed up in Figs. 6 and 7. Most of the results rely on the notion of modular phase embeddings. However, as can be easily observed by the reader, they may be extended directly to strong modular phased embeddings. Indeed, given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$, if $\mathcal{L}_1 \not\leq \mathcal{L}_2$ then $\mathcal{L}_1 \not\leq_s \mathcal{L}_2$. Moreover, as easily checked, the coders and decoders proposed to establish embeddings preserve parallel computations without interaction and are thus strong modular phased embeddings.

On a point of graphical representation, languages on the same node of the graphs embed each other in a strong modular phased way. An arrow from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ means that $\mathcal{L}_2$ strictly embeds $\mathcal{L}_1$ in a strong modular phased manner, that is $\mathcal{L}_1 \leq_s \mathcal{L}_2$, but that the converse modular phased embedding relation does not hold, $\mathcal{L}_2 \not\leq \mathcal{L}_1$. Except for the relations directly induced by the transitivity of the embeddings, the absence of an arrow between two languages means that there is no modular phased embedding, and a fortiori no strong modular phased embedding, between these languages.

In Fig. 6, the dashed arrow indicates that only the absence of strong modular phased embedding as been proved. One thus has $\mathcal{R}(ask, nask, get, tell) \not\leq_s \mathcal{R}(nask, get, tell)$. Its consequence $\mathcal{R}(ask, nask, tell) \not\leq_s \mathcal{R}(nask, get, tell)$ is only valid in a strong way. This is expressed by the dotted arrow.

Similarly, in Fig. 7, the dotted arrow means that only the relation $\mathcal{I}(ask, tell) \not\leq_s \mathcal{I}(nask, get, tell)$ has been established. However, its consequence $\mathcal{I}(ask, tell) \not\leq \mathcal{I}(get, tell)$ has been proved in a phased modular way.

One super-language $\mathcal{S}$ has been constructed from the relative time families $\mathcal{D}$ and $\mathcal{R}$. It has been proved to embed the most expressive languages of these families.

In order to compare the languages including different conceptions of time, a function $\nu$ providing current time and the notion of translation have been introduced. The existence of a translation from $\mathcal{S}_\nu$, the most expressive relative time language extended by $\nu$, into $\mathcal{I}_\nu$, the most expressive absolute time language extended by $\nu$, has been established.

Based on the expressiveness results, an implementation has been presented for $\mathcal{I}_\nu$. As a consequence of our developments, it allows for the implementation of all the languages presented in this paper.

The definition of phased embedding introduced in Section 3.2 resembles the definitions of bisimilarity relations classically defined in process algebra. In future work, we plan to define process algebras for the timed coordination languages studied in this paper, and investigate how comparison techniques used in [5,6,17] can be employed in our timed coordination context and how the proofs relate to those of this paper.

## References


[1] L. Aceto, D. Murphy, Timing and causality in process algebra, Acta Informatica 33 (4) (1995) 317–350.
[2] J.-M. Andreoli, R. Pareschi, Linear objects: Logical processes with built-in inheritance, New Generation Computing 9 (3–4) (1991) 445–473.
[3] F. Arbab, I. Herman, P. Spilling, An overview of manifold and its implementation, Concurrency: practice and experience 5 (1) (1993) 23–70.
[4] J. Baeten, J. Bergstra, Real time process algebra, Formal Aspect of Computing 3 (2) (1991) 142–188.
[5] J. Baeten, C. Middelburg, Process Algebra with Timing, Springer Verlag, 2002.
[6] J. Baeten, M.A. Reniers, Timed process algebra with a focus on explicit termination and relative timing, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, in: Lecture Notes in Computer Science, vol. 3185, Springer Verlag, 2004, pp. 59–97.
[7] J. Baeten, W. Weijland, Process Algebra, in: Cambridge Tracts in Theoretical Computer Science, vol. 18, Cambridge University Press, 1990.
[8] J. Banatre, D. LeMetayer, Programming by multiset transformation, Communications of the ACM 36 (1) (1991) 98–111.
[9] G. Berry, G. Gonthier, The Esterel synchronous programming language: Design, semantics, implementation, Science of Computer Programming 19 (1992).
[10] F.S. De Boer, M. Gabbrielli, M.C. Meo, A timed concurrent constraint language, Information and Computation 161 (1) (2000) 45–83.
[11] E. Boerger, The ASM refinement method, Formal Aspects of Computing 15 (2–3) (2003) 1–21.
[12] E. Boerger, R. Stark, Abstract State Machine: a Method for High-Level System Design and Analysis, Springer, 2003.
[13] K. De Bosschere, J.-M. Jacquet, $\mu^2 Log$: Towards remote coordination, in: P. Ciancarini, C. Hankin (Eds.), Proceedings of the Coordination Conference, in: Lecture Notes in Computer Science, vol. 1061, Springer-Verlag, April 1996, pp. 142–159.
[14] A. Brogi, P. Ciancarini, The concurrent language Shared Prolog, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 99–123.
[15] N. Busi, R. Gorrieri, G. Zavattaro, Process calculi for coordination: From Linda to JavaSpaces, in: Proc. AMAST, in: Lecture Notes in Computer Science, Springer Verlag, 2000.
[16] N. Busi, G. Zavattaro, Expired data collection in shared dataspaces, Theoretical Computer Science 298 (2003) 529–556.
[17] D. Cacciagrano, F. Corradini, Expressiveness of timed events and timed languages, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, in: Lecture Notes in Computer Science, vol. 3185, Springer Verlag, 2004, pp. 98–131.
[18] N. Carriero, D. Gelernter, Linda in context, Communications of the ACM 32 (4) (1989) 444–458.
[19] P. Caspi, N. Halbwachs, P. Pilaud, J. Plaice, Lustre: a declarative language for programming synchronous systems, in: Proc. POPL'87, ACM Press, 1987.
[20] P. Ciancarini, Distributed programming with logic tuple spaces, New Generation Computing 12 (3) (1994) 251–284.
[21] P. Ciancarini, D. Rossi, Jada: Coordination and communication for java agents, in: Proc. 2nd International Workshop on Mobile Object Systems, in: Lecture Notes in Computer Science, vol. 1222, Springer-Verlag, 1996, pp. 213–228.
[22] R. Cleaveland, A. Zwarico, A theory of testing for real-time, in: Proc. LICS'91, 1991, pp. 110–119.
[23] F. Corradini, M. Pistore, Closed interval process algebra versus interval process algebra, Acta Informatica 37 (2001) 467–509.
[24] F.S. de Boer, C. Palamidessi, Embedding as a tool for language comparison, Information and Computation 108 (1) (1994) 128–157.
[25] E. Freeman, S. Hupfer, K. Arnold, JavaSpaces: Principles, Patterns, and Practice, Addison-Wesley, 1999.
[26] D. Gelernter, N. Carriero, Coordination languages and their significance, Communications of the ACM 35 (2) (1992) 97–107.
[27] R. Gorrieri, M. Roccetti, E. Stancampiano, A theory of processes with durational actions, Theoretical Computer Science 140 (1) (1995) 73–94.
[28] D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (1987).
[29] M. Hennessy, T. Regan, A temporal process algebra, Information and Computation 117 (1995) 221–239.
[30] T. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[31] E. Horita, J.W. de Bakker, J.J.M.M. Rutten, Fully abstract denotational models for nonuiform concurrent languages, Information and Computation 115 (1) (1994) 125–178.
[32] I. Linden, J.-M. Jacquet, K. De Bosschere, A. Brogi, On the expressiveness of relative-timed coordination models, Electronic Notes in Theoretical Computer Science 97 (2004) 125–153.
[33] J.-M. Jacquet, K. De Bosschere, On the semantics of $\mu Log$, Future Generation Computer Systems 10 (1994) 93–135.
[34] J.-M. Jacquet, K. De Bosschere, A. Brogi, On timed coordination languages, in: A. Porto, G.-C. Roman (Eds.), Proc. 4th International Conference on Coordination Languages and Models, in: Lecture Notes in Computer Science, vol. 1906, Springer, 2000.
[35] I. Linden, J.-M. Jacquet, On the expressiveness of absolute-time coordination languages, in: R. De Nicola, G. Ferrari, G. Meredith (Eds.), Proc. 6th International Conference on Coordination Models and Languages, in: Lecture Notes in Computer Science, vol. 2949, Springer-Verlag, 2004, pp. 232–247.
[36] I. Linden, J.-M. Jacquet, K. De Bosschere, A. Brogi, On the expressiveness of timed coordination languages, Technical report, Institute of Informatics, University of Namur, 2005.

[37] F. Maraninchi, Operational and compositional semantics of synchronous automaton composition, in: Proc. Concur'92, in: Lecture Notes in Computer Science, vol. 630, Springer, 1992.
[38] R. Milner, Communication and Concurrency, in: International Series on Computer Science, Prentice Hall International, 1989.
[39] F. Moller, C. Tofts, A temporal calculus of communicating systems, in: Proceedings of Concur'90, in: Lecture Notes in Computer Science, vol. 459, Springer-Verlag, 1990, pp. 401–414.
[40] D. Gelernter, N. Carriero, L. Zuck, Bauhaus Linda, in: P. Ciancarini, O. Nierstrasz, A. Yonezawa (Eds.), Object Based Models and Languages for Concurrent Systems, in: Lecture Notes in Computer Science, vol. 924, Springer-Verlag, 1994, pp. 66–76.
[41] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: a kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering (1998).
[42] X. Nicollin, J. Sifakis, The algebra of timed processes, ATP: Theory and application, Information and Computation 114 (1994) 131–178.
[43] M. Nielsen, C. Palamidessi, F.D. Valencia, On the expressive power of temporal concurrent constraint programming languages, in: Proceedings of the 4th international ACM SIGPLAN conference on Principles and practice of declarative programming, ACM, 2002, pp. 156–167.
[44] G.A. Papadopoulos, F. Arbab, Coordination models and languages, Advances in Computers 48 (1998).
[45] J. Quemada, D. de Frutos, D. Azcorra, TIC: A timed calculus, Formal Aspects of Computing 5 (1993) 224–252.
[46] G.M. Reed, A.W.D. Roscoe, A timed model for communicating sequential processes, Theoretical Computer Science 58 (1988) 249–261.
[47] A. Rowstron, A. Wood, A set of Tuple space primitives for distributed coordination, in: Proc. 30th Hawaii International Conference on System Sciences, vol. 1, IEEE Press, 1997, pp. 379–388.
[48] V. Saraswat, R. Jagadeesan, V. Gupta, Programming in timed concurrent constraint languages, in: B. Mayoh, E. Tougu, J. Penjam (Eds.), Computer and System Sciences, in: NATO, vol. ASI-131, Springer Verlag, 1994.
[49] V. Saraswat, R. Jagadeesan, V. Gupta, Timed default concurrent constraint programming, Journal of Symbolic Computation 11 (1996).
[50] V.A. Saraswat, Concurrent Constraint Programming Languages, The MIT Press, 1993.
[51] E.Y. Shapiro, Embeddings among concurrent programming languages, in: W.R. Cleaveland (Ed.), Proceedings of CONCUR'92, Springer-Verlag, 1992, pp. 486–503.
[52] G. Smolka, The oz programming model, in: J. Van Leuwen (Ed.), Computer Science Today, in: Lecture Notes in Computer Science, vol. 1000, Springer Verlag, 1995, pp. 324–343.
[53] S. Tini, On the expressiveness of timed concurrent constraint programming, Electronics Notes in Theoretical Computer Science (1999).
[54] R. Tolksdorf, Coordinating services in open distributed systems with LAURA, in: P. Ciancarini, C. Hankin (Eds.), Coordination'96: First International Conference on Coordination Models and Languages, in: Lecture Notes in Computer Science, vol. 1061, Springer-Verlag, 1996.
[55] P. Wyckoff, S.W. McLaughry, T.J. Lehman, D.A. Ford, TSpaces, IBM Systems Journal 37 (3) (1998).